CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 1)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals:

✓ **Primitive Data types**

  - ✓ **Integers**
  - ✓ **Floating Point Types**
  - ✓ **Characters**
  - ✓ **Boolean**

Introduction

- Most fundamental elements of Java:
  - data types
  - variables
  - arrays

Introduction(contd.)

- **Java Is a Strongly Typed Language**
  - First, every **variable** has a **type**, every **expression** has a **type**, and every **type is strictly defined**.
  - Second, **all assignments**, whether explicit or via parameter passing in method calls, are **checked for type compatibility**.
  - **No automatic coercions or conversions** of conflicting types .
  - The Java compiler checks all expressions and parameters to ensure that the types are compatible.
  - Any type mismatches are errors that must be corrected before the compiler will finish compiling the class

The Primitive Types

- The primitive types are also commonly referred to as ***simple types*.**

- The primitive types represent **single values**—not complex objects

The Primitive Types(contd.)

Java defines eight *primitive types of data:*

- **byte**
- **short**
- **int**
- **long**

- **float**
- **double**

- **char**

- **boolean**

The Primitive Types(contd.)

Java defines eight *primitive types of data*- ***FOUR GROUPS***:

- **byte**
- **short**
- **int**
- **long**

INTEGERS

- **float**
- **double**

FLOATING-POINT NUMBERS

- **char** → **CHARACTERS**

- **boolean** → **BOOLEAN**

Primitive Types -four groups

- **Integers** This group includes **byte, short, int, and long,** which are for **whole-valued signed** numbers.
- **Floating-point numbers** This group includes **float and double,** which represent **numbers with fractional precision**.
- **Characters** This group includes **char, which represents symbols in a character set,** like **letters** and **numbers**.
- **Boolean** This group includes **boolean, which is a special type for representing true / false** values.

Integers

- Java defines four integer types:
  - **byte**
  - **short**
  - **int**
  - **long**
- Can be **signed, positive or negative values.**
- Java *does not support unsigned*, positive-only integers.
- The **width** of an integer type is not the amount of storage it consumes, but it is the behavior it defines for variables and expressions of that type

Integers

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

- The smallest integer type is byte.
- This is a signed 8-bit type
- It has a range from –128 to 127.
- Useful when working with a stream of data from a network or file.
- E.g. **declares** two byte variables called b and c:

byte b, c;

short

- short is a signed 16-bit type.
- It has a range from –32,768 to 32,767.
- It is the least-used Java type.
- Examples of **short variable declarations:**

```
short s;
short t;
```

int

- Variables of type **int are** commonly employed
  - to control loops
  - to index arrays.
- When **byte** and **short** values are ***used in an expression*** they are promoted to int when the expression is evaluated.
- int is often **the best choice** when an integer is needed.

long

- long is a signed 64-bit type and is useful for those occasions where an *int type is not large enough* to hold the desired value.
- The range of a long is **quite large**.

Floating-Point Types

- Floating-point numbers, also known as **real numbers.**
- They are used when evaluating expressions that require fractional precision.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float

- The type float specifies a **single-precision value** that uses **32 bits** of storage.
- Single precision is faster on some processors and **takes half as much space as double precision,** but will become **imprecise** when the values are either very large or very small.
- Variables of type float are useful when you need a fractional component, but **don't require a large degree of precision**.
- Example **float variable declarations:**

float hightemp, lowtemp;

double

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.
- Double precision **is actually faster** than single precision on some modern processors.
- math functions, such as **sin( ), cos( ), and sqrt( ),** return **double** values.

E.g. double

```
// Compute the area of a circle.
class Area {
    public static void main(String args[])
    {
     double pi, r, a;
     r = 10.8;
     pi = 3.1416;
     a = pi * r * r;
     System.out.println("Area of circle is " + a);
    }
}
```

OUTPUT

Area of circle is 366.436224

Characters

- In Java, the data type used to **store characters** is **char.**
- char in Java is **not the same** as char in C or C++.
  - In C/C++, char is 8 bits wide.
- Java uses **Unicode** to represent characters.
- Unicode defines a **fully international character set** that can represent all of the characters found in all human languages.
- So it requires 16 bits.
- The range of a char is **0 to 65,536.**
- There are **no negative chars**

char ch1='a';

```
// Demonstrate char data type.
class CharDemo
{
    public static void main(String args[])
    {
    char ch1, ch2;
    ch1 = 88;           // code for X
    ch2 = 'Y';
    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
    }
}
```

OUTPUT

ch1 and ch2:X Y

char act as integer type -arithmetic operations

// char variables behave like integers.

```
class CharDemo2
{
    public static void main(String args[])
    {
    char ch1;
    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);
    ch1++;                  // increment ch1
    System.out.println("ch1 is now " + ch1);
    }
}
```

OUTPUT

ch1 contains X

ch1 is now Y

Booleans

- Java has a primitive type, called **boolean**, for **logical values.**
- It can have only one of two possible **values**, **true** or **false**.
- This is the **type returned by all relational operators**,
  - boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

```
// Demonstrate boolean values.
class BoolTest
{
    public static void main(String args[]) {
    boolean b;
    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);
    if(b)
        System.out.println("This is executed.");
    b = false;
    if(b)
        System.out.println("This is not executed.");
    System.out.println("10 > 9 is " + (10 > 9));
    } }
```

OUTPUT

```
b is false
b is true
This is executed.
10 > 9 is true
```

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 2)**

Prepared by Renetha J.B.

Topics

- Literals
- Variables
- Type Conversion and Casting
- Arrays
- Strings
- Vector class.

Literals

- A **constant value** in Java is created by using a *literal* representation.
  - **Integer Literals**
  - **Floating-Point Literals**
  - **Boolean Literals**
  - **Character Literals**
  - **String Literals**

Integer Literals

- Any whole number value is an integer literal.
- Examples are 1, 2, 3, and 42
- There are three bases which can be used in integer literals
  - **Decimal(base 10)**
  - **octal (base 8)**
  - **hexadecimal (base 16).**

Integer Literals

- Normal decimal numbers
  - **cannot** have a leading zero.
  - can use digits from 0 to 9
- Octal values
  - are denoted by a leading zero.
  - can use digits from 0 to 7
  - E.g 012, 0356
- Hexadecimal constant
  - are denoted with a leading zero-x, (**0x or 0X).**
  - use digits from 0 to 9 and letters *A* through *F (or a through f)* **E.g. 0x234, 0X3B5c**

Integer Literals

- An integer literal can always be assigned to a **long variable.**
  - Append an upper- or lowercase *L to the literal*
    - 9223372036854775807L
- integer can also be assigned to a **char** as long as it is within range.
- literal value is assigned to a **byte** or **short variable as** long as it is within range.

Floating-Point Literals

- Floating-point numbers represent ***decimal values with a fractional component.***

- **Standard notation** consists of a **whole number** component followed by a **decimal point** followed by a **fractional component.**
  - **E.g.** 3.14159, 2.0
- **Scientific notation** uses a **standard-notation floating-point number** plus a *suffix (that specifies a power of 10 by which the number is to be multiplied.)*
  - The exponent is indicated by an *E or e followed by a* decimal number, which can be positive or negative
  - **E.g.** 6.022E23, 314159E–05, 2e+100.

Floating-Point Literals

- Floating-point literals in Java are **double precision by default.**
- To specify a **float** literal, we must append an **F** or **f** to the constant.
- We can also explicitly specify a **double** literal by appending a **D** or **d**.
- The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits

Boolean Literals

- Boolean literals are simple.
- There are only two logical values that a boolean value can have,
  - **true** , **false**.
- The values of true and false do not convert into any numerical representation.
- The **true** literal in Java ***does not equal 1***
- The **false** literal in Java ***does not equal 0.***

Character Literals

- Characters in Java are **indices into the Unicode character set**.
- They are 16-bit values that <u>can be converted into integers</u>
  – and manipulated with the integer operators, such as the addition and subtraction operators.
- A literal character is represented inside a pair of single quotes.
  – All of the visible ASCII characters can be directly entered inside the quotes, such as *'a', 'z', and '@'*.

Character Literals

- **‘\n’ for** the newline character.
- ‘\’’ for the single-quote character
- For octal notation, use the backslash followed by the three-digit number.
  - For example, *‘141’ is the letter ‘a’*.
- *For hexadecimal, you enter a backslash-u (**\u**), **then*** exactly four hexadecimal digits.
  - *u0061’*

String Literals

- String literals in Java are specified like they are in most other languages—by **enclosing** a sequence of characters **between a pair of double quotes**
- Examples of string literals are
  - “Hello World”
  - “two\nlines”
  - “\”This is in quotes\””

Variables

- The variable is the **basic unit of storage** in a Java program.
- A variable is defined by
  - the combination of an **identifier**, a **type**, and an ***optional* initializer**.
- All variables have a **scope,**
  - which defines their **visibility**, and a **lifetime**.

Declaring a Variable

- All variables **must be declared** before they can be used.
- The basic form of a variable declaration is :

***type identifier** [[= **value**][, **identifier** [= **value**] ...] ;*

- *The type is one of Java's atomic types, or the name of a class or interface.*
- The *identifier is the name of the* variable.
- Square bracket denote that =Value is optional in declaration.

Example- variable declaration

- **int a, b, c;** *// declares three int, a, b, and c.*
- **int d = 3, e, f = 5;** *// declares three int,*

  // initializes d to 3 and f to 5.
- **byte z = 22;** // initializes z to 22
- **double pi = 3.14159;** *// declares an approximation of pi.*
- **char x = 'x';** // the variable x has the value 'x'.

Dynamic Initialization

- Java allows variables to be **initialized dynamically, using any expression** valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;
        double c = Math.sqrt(a * a + b * b);
        // Here c is dynamically initialized
        System.out.println("Hypotenuse is " + c);
}
}
```

The Scope and Lifetime of Variables

- All of the variables used have been declared at the start of the main( ) method.
- Java allows variables to be declared within any block.
  - a block begins with an opening curly brace and ended by a closing curly brace.
  - A block defines a *scope.*
    - *A block begins with{ and end with }*
- A **scope** determines ***what objects are visible to other parts of your program.***
- **Scope** also determines the **lifetime** of those objects.

The Scope and Lifetime of Variables(contd.)

- Two major scopes are
  - Scope defined by a **class**
  - Scope defined by a **method**.
- Variables **declared inside a scope** are **not visible (that is, accessible) to <u>code that is defined outside that scope</u>**.
  - Local variable

The Scope and Lifetime of variables(contd.)

- Scopes can be nested.
  - Each time you create a block of code, we are creating a new, nested scope.
  - The outer scope encloses the inner scope.
  - This means that *objects declared in the* ***outer scope*** *will be* ***visible to code within the inner scope***.

```
{//outer
    {//inner
        {//innermost
        }
    }
}
```

```
class Sample {
public static void main(String args[])
{
    int x;          // known to all code within main function
    x = 10;
    if(x == 10)
        { // start new scope
        int y = 20; // known only to this block
                        // x(OUTER SCOPE) and y both known here.
        System.out.println("x and y: " + x + " " + y);
        x = y * 2;
        }
    // y = 100; // Error! y not known here
      // x is still known here.
    System.out.println("x is " + x);
    }
}
```

The Scope and Lifetime of variables(contd.)

```
// This fragment is wrong!
count = 100; // cannot use variable before it is declared!
int count;
```

- Variables are **created** when their scope is entered, and **destroyed** when their scope is left.
  - This means that a variable **will not hold its value once it has gone out of scope.**

// Demonstrate lifetime of a variable.

- **Variable can be reinitialized** each time it enters the block in which it is declared

```
class LifeTime {
public static void main(String args[]) {
int x;
    for(x = 0; x < 2; x++)
    {
        int y = -1;        // y is initialized each time block is entered
        System.out.println("y is: " + y);    // this always prints -1
        y = 100;
        System.out.println("y is now: " + y);
    }
} }
```

OUTPUT

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

- Although blocks can be nested, you cannot **declare a variable to have the** same name as one in an **outer scope**.

```
// This program will not compile
class ScopeErr {
public static void main(String args[])
{ int bar = 1;
  {    // creates a new scope
  int bar = 2; // Compile-time error
                   //  bar already defined in outer scope!
  }
}
}
```

Type Conversion and Casting

- If the two types are **compatible**, then Java will perform the **conversion** **automatically(implicitly)**.
  - it is always possible to assign an **int value to a long variable.**
- The conversion between incompatible types are to be done **explicitly.**

Java’s Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are **compatible**.
  - The **destination** type is **larger** than the source type.

$\longleftarrow$

Destination = source

(***same*** *type or* ***larger***)

- When these two conditions are met, a ***widening conversion*** *takes place.*

Java’s Automatic Conversions(contd.)

- For **widening conversions**, the **numeric types, including integer and floating-point types**, are **compatible** with each other.
  - No automatic conversions from the **numeric types to char or boolean.**
- Java also performs an **automatic** type conversion when a literal integer constant is stored into variables of type **byte, short, long, or char.**

byte → short →int →long → float → double

WIDENING CONVERSION

SMALL----------------------------------→ LARGE

Casting **Incompatible** Types

- If we want to assign an **int value** to a **byte variable .**
  - This conversion will **not** be performed **automatically**, because a ***byte is smaller than an int.***

byte variable=integer

(small) ← (large)

- This is called ***narrowing conversion.***
- To create a conversion between two **incompatible types,** we must use a **cast**.

Casting Incompatible Types(contd.)

- A **cast** *is* simply an **explicit** **type conversion**. It has this general form:

  (***target-type***) *value*

  - *target-type specifies the* ***desired type*** *to which value is to be converted.*

  int a;

  byte b;

  b = (**byte**) a; ***//Here integer value in variable a is casted(converted) to byte type***

- If the **integer's value** is **larger** than the range of a byte, it will be reduced to modulo *(the remainder of an integer division)* by the byte's range(256).

Casting Incompatible Types(contd.)

- A different type of conversion will occur when a **floating-point value** is assigned to an **integer** type: ***truncation.***
  - If the value 1.23 is assigned to an integer, the resulting value will simply be 1.
  - ***int a=1.23;*** *// here variable a stores only 1*

    // .23 will have been truncated

Casting Incompatible Types(contd.)

- If the size of the whole **number** component is **too large to fit into the target integer type**, then that value will be reduced modulo **the target type's range**.

  E.g.

  ```
  byte b;
  int i = 257;
  b=(byte)i;
  ```

  Here byte(-128 to 127) is smaller than 257, so the value stored in b is

  257 mod 256=1

- When the large value is cast into a **byte variable**, the ***result*** is the remainder of the division of value by 256 byte → short →int →long → float → double

WIDENING CONVERSION
(AUTOMATIC / IMPLICIT)

SMALL-----------------------------------→ LARGE

double→ float →long →int → short → byte

NARROWING CONVERSION

LARGE -----------------------------------→SMALL

EXPLICIT

Automatic Type Promotion in Expressions

byte **a** = 40;

byte **b** = 50;

byte c = 100;

int d = **a** * **b** / c; // conversions may occur in expressions.

Here intermediate term **a * b** (40*50=2000) exceeds the range of its byte operands(-128 to 127) a and b.

- To handle this kind of problem, Java automatically promotes each **byte, short,** or **char** operand to **int** <u>when evaluating an expression.</u>
- So no error.
- Variable d will contain 20

Automatic promotion

byte b = 50;

b = b * 2; // **Error**! Cannot assign an int to a byte!

- In <u>expression</u> **b*2**, automatic promotion occurs . i.e. result of b*2 (50*2=100) is **promoted to integer.**
- **This result(integer value) is larger than byte** type variable b where it is to be stored**.**
  - **So ERROR is shown.**

- To **solve** this issue, **explicit conversion** is needed for result.

byte b = 50;

b = (byte)(b * 2);

NO ERROR

The Type Promotion Rules

- First, all **byte, short, and char values** are promoted to **int**.
- If one operand is a **long**, the whole expression is promoted to **long**.
- If one operand is a **float**, the entire expression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

```
class Promote {
public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
}
}
```

f * b , b is promoted to a float (result float)

i / c, c is promoted to int, and the result is of type **int**.

d * s, the value of s is promoted to double – result double

float plus an **int** is a **float.**

float minus the **double** is promoted to **double**

RESULT double

Arrays

- An array is a group of **like-typed(same type) variables** that are referred to by **a common name**.
- Arrays of **any type** can be created
- Arrays may have one or more **dimensions**.
- A specific element in an array is accessed by its **index**.
  - Index means position It starts from 0.
    - Index of first element is 0, second element is 1 etc.

Arrays

- **One-Dimensional Arrays**
  - create an array variable of the desired type.
  - **Declaration** syntax 1

type variablename[ ];

E.g. int a[];

  - **Declaration** syntax 2

type[ ] variablename;

- The following two declarations are equivalent:

int a[];

int[]a;

Here this declaration means that **a** is an array variable, but <u>no array actually exists. No space is allocated for it in memory</u>

Arrays(contd.)

- We have to link array **with an actual, physical array of integers.**
- So we must allocate space using **new** and assign it to array variable .
  - new is a special operator that allocates memory.

***variable*=new *type*[*size*];**

E.g.

int a[];

a= new int[12];

$\boxed{\text{int a[]=new int[12];}}$

- After this statement executes, variable **a** will refer to an array of 12 integers

Array

- Obtaining an array is a two-step process.
  1. First, we must **declare** a variable of the desired array type.
  2. Second, we must **allocate the memory** that will hold the array, using **new**, and assign it to the array variable

- In Java all arrays are ***dynamically allocated.***
- It is possible to combine the declaration of the array variable with the allocation.

E.g.

```
int a= new int[12];
```

```
int a[];
a= new int[12];
```

Store value in array

```
class Array {
public static void main(String args[])
  {
      int a[];
      a = new int[4];
      a[0] = 1;
      a[1] = 3;
      a[2] = 2;
      a[3]=5;
   }
   }
```

Array initilization

- Arrays can be initialized(give values) when they are declared.
- An **array initializer** is a list of ***comma-separated*** expressions surrounded by ***curly braces***.
- No need for **new** operator

```
class AutoArray {
    public static void main(String args[])
{
    int a[] = { 1,3,2,5};
}
}
```

Array(contd.)

- If you try to access elements **outside** the range of the array (negative numbers or numbers greater than the length of the array), it will cause a **run-time error**.
- E.g

```
int a[]=new int[10];
a[-3]=5;        //ERROR
a[11]=7;        //ERROR
```

ARRAY INDEX OUT OF BOUNDS

- // Average value in an array.

```
class Average {
public static void main(String args[])
{
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;
    for(i=0; i<5; i++)
        result = result + nums[i];
    System.out.println("Average is " + result / 5);
    }
}
```

Array(contd.)

int[] num1, nums2, nums3; // create three arrays

– creates three array variables num1,num2,num3 of type **int.**

- It is the same as writing

int num1[], nums2[], nums3[];

Multidimensional Arrays

- Multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each *additional index* using another set of **square brackets.**
- **E.g 2 D array** declaration

int b[][]= new int[4][5];

This allocates a 4 by 5 array and assigns it to variable **b.**

4 rows and 5 columns

Multidimensional Arrays(contd.)

- The following declarations are also equivalent:

```
char twod[][] = new char[3][4];
char[][] twod = new char[3][4];
```

Multidimensional Arrays

- When you allocate memory for a multidimensional array, you need only **specify the <u>memory for the first</u>** (leftmost) dimension.

```
int a[][] = new int[2][];

a[0] = new int[3];

a[1] = new int[3];
```

```
int a[][]= new int[2][3];
```

  – Here **a** is 2D array with two rows. First row **a[0]** has 3 columns. Second row **a[1]** has 3 columns.

```
class TwoDArray {
public static void main(String args[]) {
int a[][]= new int[2][3];
int i, j, k = 0;
for(i=0; i<2; i++)
    {
        for(j=0; j<3; j++)
        {
        a[i][j] = k;
        k++;
        }
    }
for(i=0; i<2; i++)
    { for(j=0; j<3; j++)
        {System.out.print(a[i][j] + " ");}
      System.out.println();
    }               } }
```

OUTPUT
0 1 2
3 4 5

Array(cont.)

- When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.
- E.g.

  int a[][] = new int[2][];

  a[0] = new int[1];

  a[1] = new int[2];

  – Here array **a** has 2 rows.

  – First row a[0] has 1 column.

  – Second row a[0] has 1 column.

```
class TwoDAgain {
public static void main(String args[]) {
    int a[][] = new int[2][];
    a[0] = new int[1];
    a[1] = new int[2];
    int i, j, k = 0;
    for(i=0; i<2; i++)
    {
       for(j=0; j<i+1; j++)
       {a = k;
       k++;
    }
    for(i=0; i<4; i++) {
    for(j=0; j<i+1; j++)
       {System.out.print(a[i][j] + " ");}
    System.out.println(); }
  } }
```

OUTPUT
0
1 2

Multidimensional array initialization

- Enclose each dimension's initializer(values) within its own set of curly braces.
- We can use **expressions** as well as **literal values** inside of array initializers.
- Eg.

int a[][]={ {1,2,3} , {3,4,5} } ;

```
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 }, { 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 }, { 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
    {System.out.print(m[i][j] + " ");}
System.out.println();
}
}
}
```

```
OUTPUT
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

String class

- String is a **class**.
- It can **defines an object.**
- The String type is used to **declare string variables**
- A quoted string constant(E.g. “hello”) can be assigned to a **String variable.**
- A variable of *type String* can be assigned to another variable of *type String*.
- We can use an object of type String as an argument to println( )
- E.g.

```
String str = "this is a test";
System.out.println(str);
```

Here, str is an object of type String.
It is assigned the string “this is a test”.
This string is displayed by the println( ) statement.

String E.g.

```
class Sample {
public static void main(String args[])
    {
    String s="Hello"
    System.out.print(s);
    }
}
```

OUTPUT
Hello

String(contd.)

- In Java, string is basically an **object** that represents sequence of char values .
- An array of characters works same as Java string.
- For example:

char[] ch={'H','e','l','l','o'};

String s=new String(ch);

*//This statement converts character array **ch** to string and store in string object s.*

This is same as

String s="Hello"; //creating string by java string literal

String methods

- length()
  - The length of a string can be found with the length() method.

```
class Sample {
  public static void main(String args[])
   {
   String s="Hello";
   System.out.print("Length=",s.length());
   }
}
```

OUTPUT
Length=5

String methods(contd.)

- toUpperCase() and toLowerCase()
  - To convert from lower to upper and upper to lower respectively

```
class Sample {
public static void main(String args[])
    {
    String s="Hello World";
    System.out.println(s. toUpperCase());
    System.out.println(s. toLowerCase());
    }
}
```

OUTPUT
HELLO WORLD
hello world

String methods(contd.)

- indexOf()
  - The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace)

```
class Sample {
public static void main(String args[])
    {
    String s="I am fine.I am ok";
    System.out.println(s.indexOf("am"));
    }
}
```

OUTPUT
2

String concatenation

- <u>Method 1:</u>The + operator can be used between strings to combine them. This is called concatenation
- <u>Method 2:</u>We can use **concat()** method to concatenate two strings.

```
class Sample {
public static void main(String args[])
    {
    String s1="Computer" , s2="Science"
    System.out.println(s1+s2);
    }
}
```

OUTPUT
ComputerScience

String concatenation(contd.)

```
class Sample {
public static void main(String args[])
      {
      String s1="Computer " , s2="Science";
      System.out.println(s1+s2);
      }
}
```

OUTPUT
Computer Science

String concatenation(contd.)

- If we add a number and a string, the result will be a string concatenation.

```
class Sample {
public static void main(String args[])
    {
    String s1="10 " , s2="12";
    int a=13;
    System.out.println(s1+s2);
    System.out.println(s1+a);

    }
}
```

OUTPUT
1012
1013

Vector class

- Vector implements a **dynamic array**.
  - it can **grow** or **shrink** in size as required.
- It is similar to ArrayList class, but with two differences:
  - Vector is synchronized, and it contains many legacy methods that are not part of the Collections Framework.
    - Synchronized **means** if one thread is working on **Vector**, no other thread can get a hold of it.
  - Vector can **extend AbstractList class** and can **implement** the **List interface.**

Vector(contd.)

- All vectors **start with an initial capacity(size).**
- After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object **plus extra room for additional objects**.
- The amount of extra space allocated during each reallocation is determined by the ***increment*** that you specify when you create the vector.
- If we don't specify an ***increment***, **the vector's size is doubled** by each allocation cycle.

Vector(contd.)

- Vector is declared like this:

class Vector<E>

- Here, E specifies the type of element that will be stored.
- **Vector constructors** are

Vector( )

Vector(int *size)*

Vector(int *size, int incr)*

Vector(Collection<? extends E> *c)*

Vector(contd.)

- **Vector**( ) creates a **default vector**, which has an initial size of 10.
- **Vector(int *size)*** creates a vector whose initial capacity is specified by *size.*
- **Vector(int *size, int incr)*** creates a vector whose minitial capacity is specified by *size and whose increment is specified by incr.*
  - The **increment** specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector(Collection<? extends E> *c)*** *c*reates a vector that contains the elements of collection *c*.

Vector(contd.)

- Vector defines these protected data members:

int **capacityIncrement;**

int **elementCount**;

Object[ ] **elementData**;

- The increment value is stored in capacityIncrement.
- The number of elements currently in the vector is stored in elementCount.
- The array that holds the vector is stored in elementData.

Vector(contd.)

- Vector defines several legacy methods

Method	Description
void addElement(E *element*)	The object specified by *element* is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object *element*)	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt(E *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.
boolean isEmpty( )	Returns **true** if the vector is empty, and returns **false** if it contains one or more elements.
E lastElement( )	Returns the last element in the vector.

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

The Scope and Lifetime of Variables(contd.)

- The **scope defined by a method** begins with its opening curly brace. {
  - If that method has **parameters**, they ***too are included within the method's scope.***

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 3)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals:
  - ✓ **Operators –**
    - ✓ Arithmetic Operators,
    - ✓ Bitwise Operators,
    - ✓ Relational Operators,
    - ✓ Boolean Logical Operators,
    - ✓ Assignment Operator,
    - ✓ Conditional (Ternary) Operator,
    - ✓ Operator Precedence.

Operators

- Operators are used for performing operations.
  - **Arithmetic Operators**

    +, -, *, /, % ++, +=, -=,*=, /=, %=, - -
  - **Bitwise Operators** ~ Bitwise unary NOT

    &,| ,^ , >> ,>>> ,<< , &=, |=, ^=, >>=, >>>=
  - **Relational Operators**

    = =, !=, >, < , <=, >=
  - **Boolean Logical Operators**

    &, |, ^, ||,&&, &=,| =, ^=, ==, !=, ?:
  - **Assignment Operator**

    =
  - **Conditional (Ternary) Operator**

    ?:

- Assignment Operator

  =

- Conditional (Ternary) Operator

  ?:

Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Basic Arithmetic Operators

- The basic arithmetic operations—addition, subtraction, multiplication, and division works for all numeric types.
  - The minus operator also has a unary form that negates its single operand.
  - E.g

```
int a=3;
int b=-a;
```

Modulus Operator

- **The** Modulus Operator
- The modulus operator, **%, returns the remainder of a division operation. It can be applied to**
- floating-point types as well as integer types. The following example program demonstrates
- the **%:**

Arithmetic Compound Assignment Operators

- Variable operator = expression;

This is same as

Variable =Variable operator expression;

- In programming:

a = a + 4;

can be written as

a += 4;

E.g.

```
int a=3;
a+=2;      //Now value of a is 3+2=5
```

// Demonstrate the % operator.

```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program, you will get the following output:

x mod 10 = 2

y mod 10 = 2.25

Pre-Increment Post increment

- Pre increment E.g

x = 42;

y = ++x;

x 43
y 43

- Post increment E.g

x = 42;

y = x++;

x 43
y 42

Bitwise operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise logical oprators

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Examples

```
  00101010    42
& 00001111    15
----------
  00001010    10
```

```
  00101010    42
| 00001111    15
----------
  00101111    47
```

```
  00101010    42
^ 00001111    15
----------
  00100101    37
```

```
~00101010
becomes
 11010101
```

Right shift

- Each time you shift a value to the right, it divides that value by two—and discards any remainder.

- When you are shifting right, **the top (leftmost) bits** exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension and serves to preserve* the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4

```
11111000    –8
>>1
11111100    –4
```

Right shift e.g

- E.g.

```
int a = 32;
a = a >> 2; // a now contains 8
```

- E.g.

```
int a = 35;
a = a >> 2; // a still contains 8
```

```
00100011      35
>> 2
00001000       8
```

Unsigned, shift-right operator, >>>

- Shift a zero into the high-order bit(letftmost or top) no matter what its initial value was. This is known as an *unsigned shift.*
- Java's unsigned, shift-right operator, **>>>** always shifts zeros into the high-order bit.
- E.g **a is set to –1**, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
11111111 11111111 11111111 11111111   –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111   255 in binary as an int
```

Relational operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Relational operator(contd.)

int a = 4;

int b = 1;

boolean c = a < b;   //c contains false. 4 is not less than 1

Here the result of **a<b** (which is **false**) is stored in c.

E.g.

int done;

// ...

if(!done) ... // Valid in C/C++

if(done) ... // but not valid in Java.

if(done == 0) ... // This is Java-style.

if(done != 0) ... 18

Boolean Logical Operators

<table>
<tr><th>Operator</th><th>Result</th></tr>
<tr><td>&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Logical OR</td></tr>
<tr><td>^</td><td>Logical XOR (exclusive OR)</td></tr>
<tr><td>||</td><td>Short-circuit OR</td></tr>
<tr><td>&&</td><td>Short-circuit AND</td></tr>
<tr><td>!</td><td>Logical unary NOT</td></tr>
<tr><td>&=</td><td>AND assignment</td></tr>
<tr><td>|=</td><td>OR assignment</td></tr>
<tr><td>^=</td><td>XOR assignment</td></tr>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>?:</td><td>Ternary if-then-else</td></tr>
</table>

- The logical Boolean operators, **&, |, and ^, operate on boolean values in the same way** that they operate on the bits of an integer.

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Short-Circuit Logical Operators

- Secondary versions of the Boolean AND and OR operators, and are known as ***short-circuit logical operators.***
- The OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is.
- If you use the || and && forms, rather than the | and & forms of these operators, Java <u>will not bother to evaluate the right-hand operand</u> when the **outcome of the expression can be determined by the left operand alone**.

Short-Circuit Logical Operators(E.g)

- E.g

if (**denom != 0** && num / denom > 10)

- Here if denom is 0 the second expression is not validated
  - So there is no risk of causing a run-time exception when denom is zero.
- If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero.

Assignment Operator

- *var = expression;*
- Here, the type of *var must be compatible with the type of expression.*
- It allows you to create a chain of assignments

int x, y, z;

x = y = z = 100; // set x, y, and z to 100

Ternary (conditional or three-way) operator

- The ? Operator has this general form:

expression1 ? expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value.
  - If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.
  - The result of the ? operation is that of the expression evaluated.
  - Both expression2 and expression3 are required to return the **same type**, which can't be void

E.g.

- int ratio = denom == 0 **?** 0 : **num / denom**;
  - If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression.
    - Here 0 is stored in ratio
  - If denom does **not equal zero**, then the expression after the colon is evaluated and used for the value of the entire ? expression.
  - i.e num/denom is stored in ratio
- The result produced by the ? operator is then assigned to ratio.

```
int a=3,b=5;
int c=(a>b?a:b);
```

- Here a>b is **false** so the value of b is stored in c.

Operator Precedence

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

Associativity of operators

- When an expression has two or more operators with the same precedence, the expression is evaluated according to its **associativity**.
  - It is the order of applying operators

Operator Associativity

Operator	Type	Associativity
() [] .	Parentheses Array subscript Member selection	Left to Right
++ --	Unary post-increment Unary post-decrement	Right to left
++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
* / %	Multiplication Division Modulus	Left to right
+ -	Addition Subtraction	Left to right

<table>
<tr><td><<
>>
>>></td><td>Bitwise left shift
Bitwise right shift with sign extension
Bitwise right shift with zero extension</td><td>Left to right</td></tr>
<tr><td><
<=
>
>=
instanceof</td><td>Relational less than
Relational less than or equal
Relational greater than
Relational greater than or equal
Type comparison (objects only)</td><td>Left to right</td></tr>
<tr><td>==
!=</td><td>Relational is equal to
Relational is not equal to</td><td>Left to right</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>Left to right</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>Left to right</td></tr>
<tr><td>|</td><td>Bitwise inclusive OR</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to right</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to right</td></tr>
<tr><td>? :</td><td>Ternary conditional</td><td>Right to left</td></tr>
<tr><td>=
+=
-=
*=
/=
%=</td><td>Assignment
Addition assignment
Subtraction assignment
Multiplication assignment
Division assignment
Modulus assignment</td><td>Right to left</td></tr>
</table>

Associativity

- Right to Left associative
  - Unary operators
  - Assignment operators
  - Conditional(ternary) operators)
- All other operators are Left to Right associative

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 4)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals**:**

✓ **Control Statements**

  ✓ Selection Statements,

  ✓ Iteration Statements

  ✓ Jump Statements.

Control statements

- A programming language uses control statements to cause the **<u>flow of execution</u>** <u>to</u> **<u>advance</u>** <u>and</u> **<u>branch</u>** <u>based on changes to the state of a program</u>
- Categories of control statements
  - ✓ Selection Statements,
  - ✓ Iteration Statements
  - ✓ Jump Statements.

Control statements(contd.)

- **Selection statements** allow the program
  - to choose different paths of execution based on condition (outcome of an expression or the state of a variable).
- **Iteration statements** enable program execution
  - to *repeat one or more statements* (that is, iteration statements form loops).
- **Jump statements** allow your program
  - to execute in a nonlinear fashion.

Java's Selection Statements

- Also called **decision making statements.**
- Selection statements **control the flow of program's execution** based upon **conditions** *known only during run time.* It helps to choose different paths of execution based on condition.
- Java supports two selection statements:
  - ✓ **if**
  - ✓ **switch**

if statement

- **if** statement is Java’s **conditional branch statement.**
  - It can be used to route program execution through different paths.

- Syntax of **simple if** statement

```
if (condition)
{
 // block of code to be executed if the condition is true
………………
}
```

Simple if E.g.

```
class Sample{
public static void main(String args[])
{
    int a=5;
    if(a>0)
    {
      System.out.println(" a is a positive number");
    }
}
```

If-else statement

- General form of the **if statement:**

if (***condition***) *statement1;*

else *statement2;*

- Statement may be **a single statement** or a **compound statement enclosed in curly braces** (that is, a block).
- The ***condition*** is any expression that returns a **boolean value.**
- The **else** clause is optional.

Working of if-else

if (***condition***) *statement1;*

else *statement2;*

- If the condition is true, then *statement1* is executed.
- Otherwise, *statement2* (if it exists) is executed.
- Both statements will not be executed at the same time.

If-else E.g

```
class Sample{
public static void main(String args[]) {
int a=5, b=3;
if(a < b) a = 0;
else b = 0;
System.out.println(" a=" + a);
System.out.println(" b=" + b);
}
}
```

OUTPUT
a=5
b=0

If statement(contd.0

- If statement can be controlled using a boolean variable.
- E.g.

```
…
boolean dataAvailable;
// ...
if (dataAvailable)                //if dataAvailable is true
    ProcessData();                //call this function
else
    waitForMoreData();            //call this function
..
```

If statement(contd.)

- Only one statement can appear directly after the **if** or the **else.**

```
if(condition)
Statement1;
else
Statement
```

- If we want to include **more statements** inside **if** statement or **else** , we have to create a block (start with **{** and end with**}**

```
if(condition)
{
Statement1;
Statement1;
.....
}
```

Nested ifs

- A **nested if** is an if statement that is the inside (target of) another **if** or **else**.
- The **else statement** always refers to the
  - nearest if statement that is within the **same block** as the else and that is *not already associated with an else*.

Nested if E.g

```
if(i == 10)
    {
        if(j < 20) a = b;
            if(k > 100) c = d;      // this if is
            else a = c;             // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder

- A common programming construct that is based upon a **sequence of nested ifs** is the **<u>if-else-if ladder.</u>**

if(*condition)*

statement;

else if(*condition)*

statement;

else if(*condition)*

statement;

...

else

statement;

The if-else-if Ladder(contd.)

- The if statements are executed from the **top down**.
- As soon as one of the conditions controlling the **if is true**, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If **NONE of the conditions is true**, then the final else statement will be executed.
- The last **else** acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");}}
```

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");}}
```

switch statement

- The switch statement is Java's **multiway branch** statement.
- It is an better alternative than a large series of **if-else-if** statements.

Syntax of switch

```
switch (expression)
{
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
    break;
    ...
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

Switch(contd.)

switch(***expression***){…..}

- The ***expression* inside switch** must be of type byte, short, int, or char;
  - each of the values specified in the case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement)

Working of switch

- The value of the expression inside switch is compared with each of the literal values in the case statements.
  - If a match is found, the code sequence following that case statement is executed.
  - If none of the constants in the case matches the value of the expression, then the default statement is executed.
  - default statement is optional.
  - If no case matches and no default is present, then no further action is taken.

Working of switch(cond.)

- The break statement is used inside the switch to <u>terminate a statement sequence.</u>
- When a break statement is encountered, <u>execution branches to the first line of code **after the entire switch** statement.</u>
- This has the effect of "jumping out" of the switch.

```java
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<5; i++)
    switch(i)
    {
    case 0:
        System.out.println("i is zero.");
        break;
    case 1:
        System.out.println("i is one.");
        break;
    case 2:
        System.out.println("i is two.");
        break;
    default:
      System.out.println("i is greater than 2.");
    } } }
```

OUTPUT

```
i is zero.
i is one.
i is two.
i is greater than 2.
i is greater than 2.
```

```
class Switcheg {
public static void main(String args[]) {
for(int i=0; i<4; i++)
switch(i)
{
case 0:
    System.out.println("i is zero.");
case 1:
    System.out.println("i is one.");
case 2:
    System.out.println("i is two.");
    break;
default:
    System.out.println("i is greater than 2.");
} } }
```

OUTPUT

i is zero.
i is one.
i is two.
i is one.
i is two.
i is two.
i is greater than 2.

```
import java.util.Scanner;
class Switchvow {
public static void main(String args[]) {
Scanner sc=new Scanner(System.in);
System.out.println("Enter a letter:");
char c=sc.next().charAt(0); ;
switch(c)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
case 'A':
case 'E':
case 'I':
case 'O':
case 'U':
System.out.println("vowel");
break;
default:  System.out.println("Not vowel-may be consonent");  } } }
```

OUTPUT
Enter a letter:
A
vowel

Nested switch Statements

- We can use a switch as part of the statement **sequence of an outer switch**. This is called a *nested switch*

```
switch(count) {
    case 1:
        switch(target) { // nested switch
                case 0:
                        System.out.println("target is zero");
                        break;
                case 1:   // no conflicts with outer switch
                        System.out.println("target is one");
                        break;
                }
        break;
    case 2: // ...   }
```

Features of the switch statement

- The switch differs from the if in that **switch can only test for equality**, whereas if can evaluate any type of Boolean expression.
  - switch looks only for a match between the value of the expression inside **switch** and one of its **case** constants.
- No two case constants in the same switch can have identical values.
  - But a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually **more efficient** than a set of nested ifs.

Swich(features)

- When **Java compiler** compiles a switch statement, it will inspect each of the case constants and <u>create a “jump table”</u> that it will use for selecting the path of execution depending on the value of the expression.
- So a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses.
- The compiler can do this because it knows that the case constants are all the same type and simply must be **compared for equality** with the switch expression.
- The compiler has no such knowledge of a long list of if expressions

Iteration Statements

- A iteration statements or loop repeatedly executes the same set of instructions until a termination condition is met.

- Java's iteration statements (**looping** statements) are
- ✓ **for**
- ✓ **while**
- ✓ **do-while**

while

- The while loop is Java's most fundamental loop statement. It is **ENTRY CONTROLLED** loop.
  - The statements inside the body of **while** is executed only if the *condition inside while is **true***.
- It repeats a statement or block while its controlling expression is true.
- General form:

```
while(condition)
{
// body of loop
}
```

Working of while

```
while(condition)
{
// body of loop
}
```

- The **condition** can be any Boolean expression.
  - The body of the loop will be executed as long as the conditional expression is **true**.
  - When condition becomes **false**, control passes to the next line of code immediately after the loop.

While(contd.)

- The curly braces are not needed if only a single statement is being repeated.

```
while(condition)
    Statement;
```

```
// Demonstrate the while loop.
class Whileeg {
public static void main(String args[]) {
int n = 10;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

OUTPUT

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

While(contd.)

- The body of the while (or any other of Java's loops) can be empty.
  - This is because a **null statement** (one that consists only of a **semicolon**) is syntactically valid in Java.

while(condition) ;

Here if condition is true no statement is executed as part of while

```
class Whileeg {
public static void main(String args[])
{
    int n = 10;
    while(n > 0)
     {
    System.out.println("tick " + n);
     }
}
}
```

OUTPUT
tick 10
tick 10
tick 10
…..

.

.
INFINITE LOOP

do-while

- The **do-while loop always executes its body at least** once, because its conditional expression is at the bottom of the loop.

```
do
{
   //statements
}
while(condition);
```

Working of do-while

do-while is EXIT CONTROLLED loop.

```
do
{
   //statements
}
while(condition);
```

1. Initially the **statements** inside the do-while loop is executed
2. then only the **condition** inside while is checked.
3. Then the loop is executed only if that condition is true.
   – That is condition is checked only during exit from do-while loop.

```
class Menu {
public static void main(String args[]) throws java.io.IOException
    {
...
char choice = (char) System.in.read();
…
}
```

Here **System.in.read( )** is used here to obtain the value of choice from user. So main() function throws java.io.IOException

```
class Whileeg {
public static void main(String args[])
{
   do
    {
   System.out.println("Hello");
   }while(true);
}
}
```

OUTPUT
Hello
Hello
Hello
Hello
….
…
..
.
INFINITE LOOP

Difference between while and do-while

BASIS FOR COMPARISON	WHILE	DO-WHILE

General Form	`while ( condition) {`
	`statements; //body of loop`
	`}`
		`do{`
		`.`
		`statements; // body of loop.`
		`.`
		`} while( Condition );`

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

for

It is an iteration statement(looping)

for(***initialization***; *condition;* ***iteration***) *{*

// body

}

Working of for loop

- When the loop first starts, the **initialization** portion of the loop is executed. It acts as a loop control variable (counter).
  - the initialization expression is only executed once.
- Next, condition is evaluated.(Boolean expression)
  - It usually tests the loop control variable against a target value.
  - If this expression is **true**, then the body of the loop is executed.
  - If it is **false**, the loop terminates.
- Next, the iteration portion of the loop is executed.
  - increments or decrements the loop control variable.
- Next, condition is evaluated.
- And the process continues until condition becomes **false**

```
for( ; ; )
{
// ...
}
```

INFINITE LOOP

The For-Each Version of the for Loop

for(*type var :* collection) statement-block;

- Here, *type* specifies the type and *var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection

```
class Foreacheg {
public static void main(String args[])
{
 int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x: nums)
    System.out.println(x);
}
}
```

OUTPUT
1
2
3
4
5
6
7
8
9
10

- During each pass through the loop, x is automatically given a value equal to the next element in nums.
  - Thus, on the first iteration, x contains 1;
  - on the second iteration, x contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
    System.out.println(x);
```

```
1
2
3
4
5
6
7
8
9
10
```

- With each pass through the loop, x is automatically given a value equal to the next element in nums.
  - Thus, on the first iteration, x contains 1;
  - on the second iteration, x contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

Nested loops

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print("i="+i+" j="+j + "\t\t");
System.out.println();
}
}
}
```

OUTPUT

```
i=0 j=0        i=0 j=1        i=0 j=2
i=1 j=0        i=1 j=1        i=1 j=2
i=2 j=0        i=2 j=1        i=2 j=2
i=3 j=0        i=3 j=1        i=3 j=2
```

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print(i +"\t\t");
System.out.println();
}
}
}
```

OUTPUT

0	0	0
1	1	1
2	2	2
3	3	3

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print(j+"\t\t");
System.out.println();
}
}
}
```

OUTPUT

0	1	2
0	1	2
0	1	2
0	1	2

Nested loops

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<2; j++)
System.out.print(“*");
System.out.println();
}
}
}
```

```
**
**
**
**


.
```

Nested loops

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=i; j<4; j++)
System.out.print("*");
System.out.println();
}
}
}
```

```
****
***
**
*

.
```

Jump Statements

- ❑ Java supports three jump statements:
- ✓ **break**
- ✓ **continue**
- ✓ **return**

break statement

- Three uses.

✓ First it **terminates** a statement sequence in a switch statement.

✓ Second, it can be used to **exit** a loop.

✓ Third, it can be used as a "civilized" form of goto.

break E.g

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<6; i++)
{
if(i == 3)
    break; // terminate loop if i is 3
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

OUTPUT

```
i: 0
i: 1
i: 2
Loop complete.
```

Using break as a Form of Goto

- By using this form of **break, you can, for example, break out of** one or more blocks of code.
- The general form of the labeled break statement is :

  break *label;*

```
// Using break as a civilized form of goto.
class Breakeg {
public static void main(String args[]) {
boolean t = true;
first: {
        second: {
                third: {
                        System.out.println("Before the break.");
                        if(t) break second; //break of second block
                        System.out.println("This won't execute");
                        }
                System.out.println("This won't execute");
                }
   System.out.println("After second block.");
   }
}
}
```

OUTPUT
Before the break.
After second block..

continue statement

- In **while** and **do-while** loops, a **continue** statement causes **control to be transferred directly to the conditional expression** that controls the loop.
- In a **for** loop, **control goes first to the iteration portion of the for statement and then to the conditional expression**.
- For all three loops, **any intermediate code after continue is bypassed(skipped).**

continue E.g

```
// Using break to exit a loop.
class continueeg{
public static void main(String args[]) {
for(int i=0; i<6; i++)
{
if(i == 3)
    continue; // skip remaining stmts if i is 3
        // continue loop.control goes to iteration
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

OUTPUT
i: 0
i: 1
i: 2
i: 4
i: 5
Loop complete.

```java
// Demonstrate continue.
class Continueeg {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

OUTPUT

```
0 1
2 3
4 5
6 7
8 9
```

return statement

- The **return** statement is used to explicitly return from a method.
  – The **return** causes program control to transfer back to the caller of the method.
- When **return** statement is executed the method terminates.
- The **return** causes execution to return to the Java run-time system
- Methods that have a <u>return type other than void</u> **return a value** to the calling method(function)

return *value;*

  – Here, *value is the value is returned to the calling function*

```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return;
System.out.println("This won't execute.");
}
}
```

OUTPUT
Before the return

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 5)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals**:**

✓ **Object Oriented Programming in Java**

  - **Class Fundamentals**
  - **Declaring Objects**
  - **Object Reference**
  - **Introduction to Methods.**

Class Fundamentals

- The **class** is the core of Java.
  - The class forms the basis for object-oriented programming in Java.
- A class is a "blueprint" for creating objects
- A **class** is a ***template for an object.***
  - An **object** is an ***instance of a class.***
- A class defines a *new type of data*.
- A class creates a ***logical framework*** that defines the relationship between its members.

Example

Student

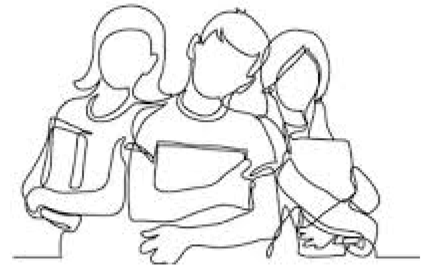
(class)

Rollno
Name

read()
write()

properties
(instance variable:

behaviour
(methods)

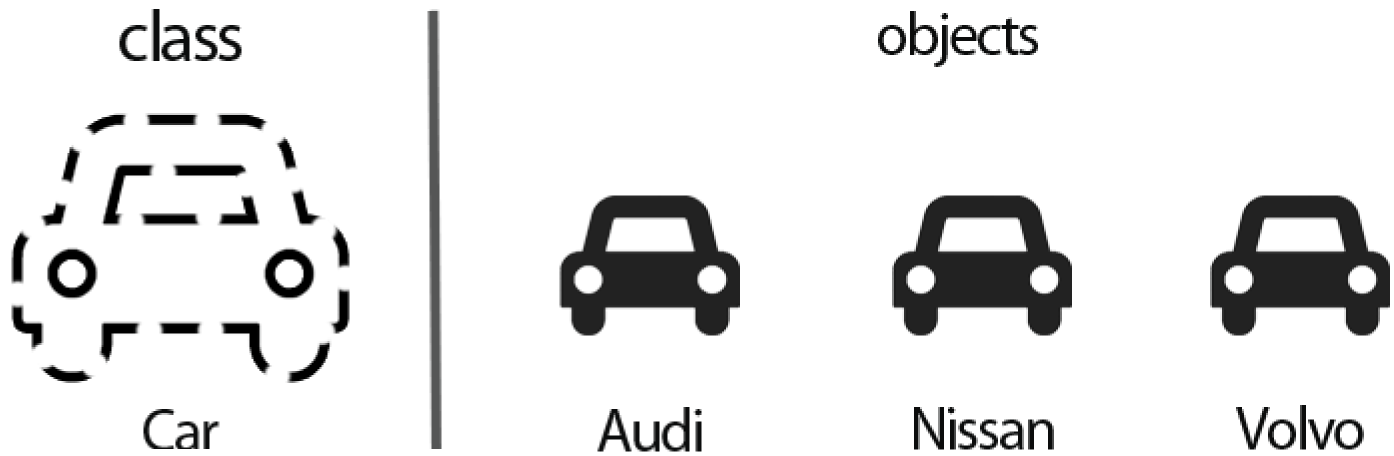


12
Smith

object

2
Susan

object



model   **colour**   **price**
start()   **brake()**   **reverse()**   **stop()**

properties
(instance variables)

behaviour
(methods)

Class(continued.)

- A class is declared using the keyword **class**
- The data or variables, defined within a class are called **instance variables**.
  - because each instance of the class (that is, each object of the class) contains its own copy of these variables.
  - the data for each object is separate and unique.
- Functions inside class are called **methods**.
- The methods and variables defined within a class are called **members of the class.**

The General Form of a Class

- **A** general form of a **class definition is**

class *classname*

{

type ***instance-variable1;***

type ***instance-variableN;***

Properties **(instance variables)**

type methodname1(parameter-list)

Behaviour **or** **Method** or **function**

{

// body of method

}

type methodnameN(parameter-list)

Behaviour or **Method** or **function**

{ // body of method

}

}

Members of class

A Simple Class

```
class Box
{
double width;
double height;
double depth;
}
```

Properties (instance variables) ← **Member** of class

A Simple Class

```
class Box
{
double width;
double height;
double depth;

void  volume()
{
//statements
}


}
```

Properties
(instance variables)

Behaviour or
Method or
Function

Members
of class

Declaring Objects

- When we create a class, we are creating a new data type.
  - We can use this type to declare objects of that type.
- Obtaining objects of a class is a two-step process.
  - *First*, we must **declare a variable of the class type.**
    - This variable does not define an object.
    - It is simply a variable that can *refer to an object.*
  - *Second*, we must **acquire an actual, physical copy of the object and assign it to that variable** (using **new** operator)

Declaring Objects(contd.)

Classname objectname ; *// declare reference to object*

objectname = **new** Classname(); *// allocate an object*

- We can write this in a single statement

Classname objectname = **new** Classname();

class Box

```
{double Width;
double Height;
double Depth;
}
```

Box mybox;

- This line declares **mybox** as a **reference to** an object of type **Box**.
- Here **mybox** contains the value **null**, which indicates that it does not yet point to an actual object

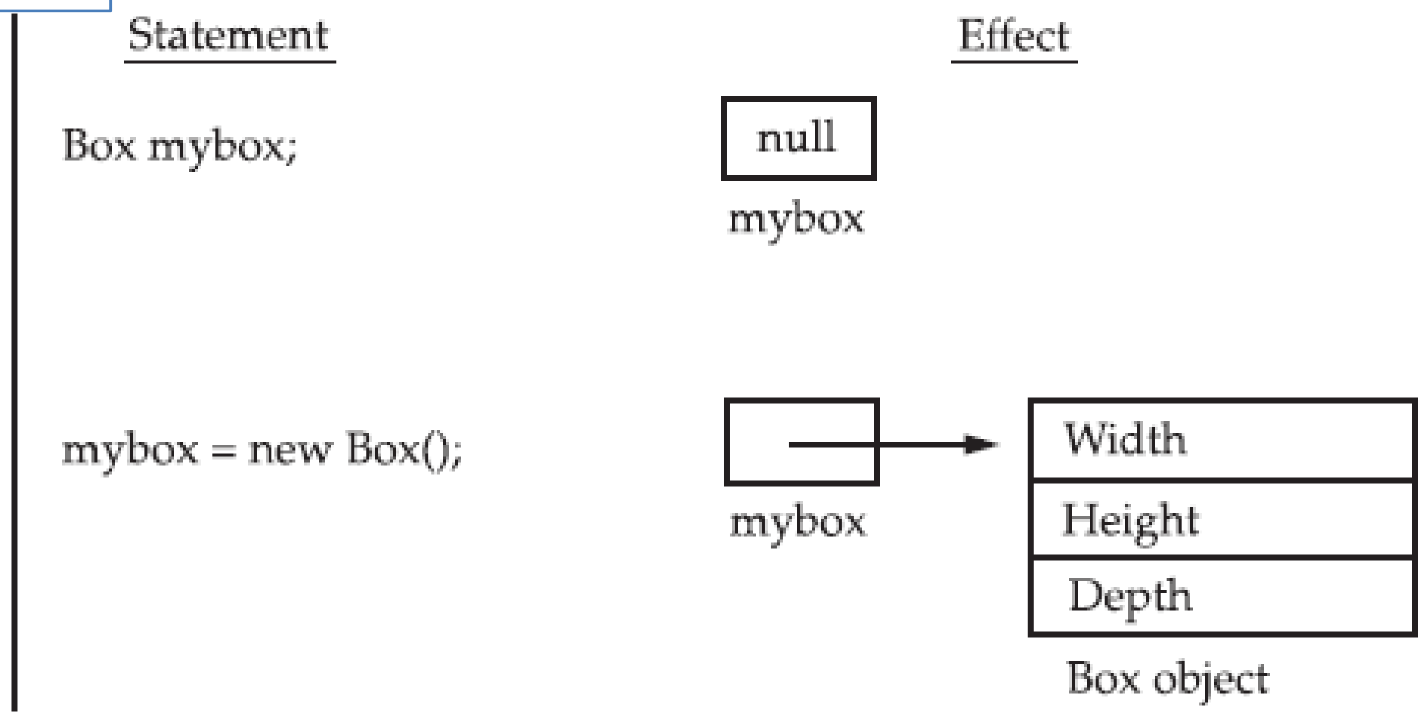
mybox = new **Box**();

- This line allocates an actual object and assigns a reference to it to **mybox.**
- **mybox** holds the memory address of the actual Box object.

Declaring Objects(contd.)

```
class Box
{double Width;
double Height;
double Depth;
}
```

Declaring an object of type **Box**



Declaring Objects(contd.)

- The class name followed by parentheses specifies the *constructor* for the class.

Box mybox=new Box();

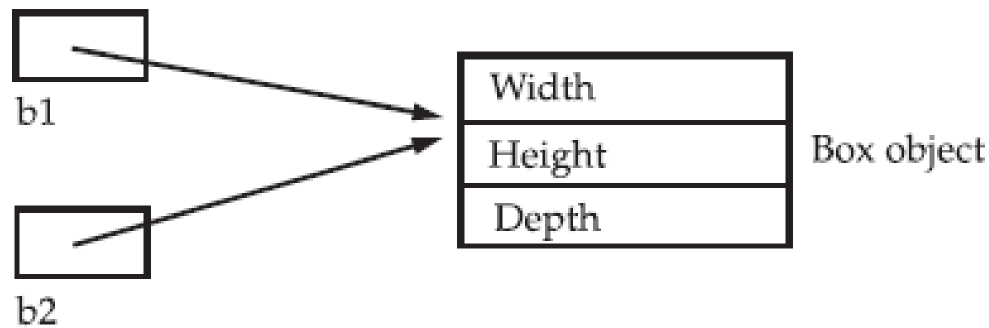
- Here Box is the class. Box()is the constructor.
- A constructor defines what occurs when an object of a class is created.

Assigning Object Reference Variables

- Object reference variables act differently when an assignment takes place
- E.g.

Box b1 = new Box();

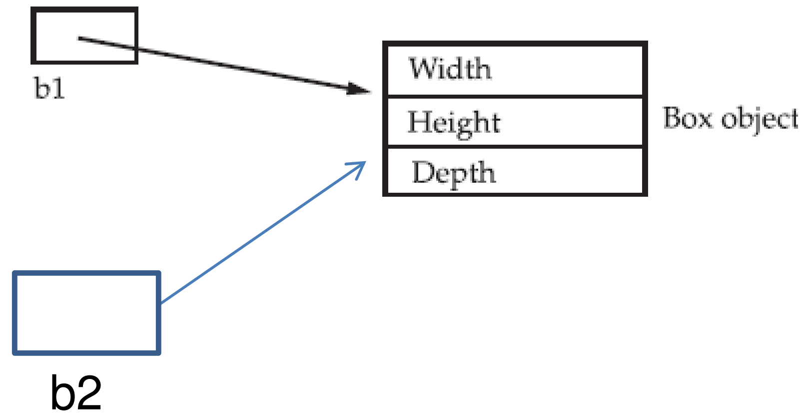
Box b2 = b1;



- Here b1 and b2 will both refer to the ***same object.***
- Any changes made to the object through b2 will affect the object which is referred by b1, because they are the same object.

Assigning Object Reference Variables (contd.)

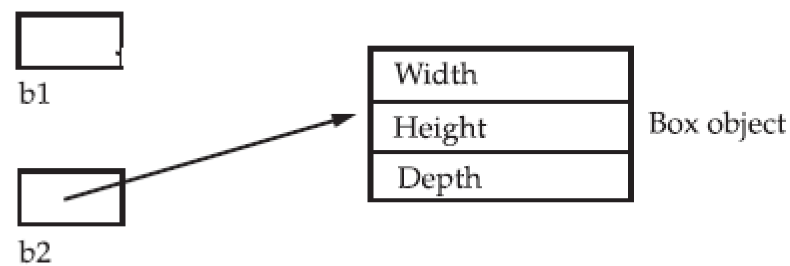
Box b1 = new Box();



Box b2 = b1;

// ...

b1 = null;



- Here at the end b1 has been set to null, but b2 still points to the original object.

Class vs object

Class

- **Template** for creating objects
- **Logical** entity
- Declared using **class** keyword
- Class **does not get any memory** when it is created.
- A class is **declared only once**

Object

- **Instance** of class
- **Physical** entity
- Created using **new** operator.
- Object **gets memory** when it is created using new operator.
- **Many objects** can be created from a class

Introducing Methods

- **Classes** usually consist of two things:
  - Instance variables
  - Methods or functions.
- The general form of a method:

type name(parameter-list)

{

// body of method

}

- The *type* specifies the type of data returned by the method.
  - any valid type, including class types, void
- The *parameter-list or argument list is a* sequence of type and identifier pairs separated by commas.

Introducing Methods(contd.)

- Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

    return value;

- Method of one class can be invoked by functions of other classes through objects of former class.

Objectname.method(parameters);

// EXAMPLE

```
class Box {
    double width;
    double length;
    double depth;

    void volume()
    {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
    }
}
```

Properties
(**instance variables**)

Behaviour or
Method or
Function

Box class

```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1. length = 30;
mybox1.depth = 15;
mybox1.volume();
} }
```

Behaviour or
Method or
Function
MAIN FUNCTION

Object of class **Box**

BoxDemo class

Prepared by Renetha J.B.

Example

- Create a class Box with instance variables length, width and height. Include a method volume to compute the volume of the box,
- Create another class BoxDemo with main function that creates an object of class Box named mybox1 and set the values for instance variables(length, width and height). Invoke the function volume in Box to compute the volume of the created object mybox1

```
class Box {
        double width;
        double length;
        double depth;

        void volume()
        {
        System.out.print("Volume is ");
        System.out.println(width * length * depth);
        }
}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1. length = 30;
mybox1.depth = 15;
mybox1.volume();
}
}
```

OUTPUT

Volume is 3000.0

```
// program using return statement
class Box {
	double width;
	double height;
	double depth;

	void volume()
	{
	return(width * length * depth);
	}
}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
int v=mybox1.volume();
System.out.println(”Volume=“+v);
} }
```

OUTPUT

Volume is 3000.0

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 6)**

Prepared by Renetha J.B.
Dept.of CSE, LMCST

Topics

- Core Java Fundamentals:
  - ✓ **Constructors**
  - ✓ **this Keyword**
  - ✓ **Method Overloading**
  - ✓ **Using Objects as Parameters**

Constructor

- A constructor **help to initialize an object**(give values) immediately upon creation.
- Constructor is a special method inside the class.
- Constructor has the same name as the class in which it resides.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

Constructor(contd.)

- Constructors have no return type, not even void.
  - This is because the implicit return type of a class' constructor is the class type itself.
- Two types of constructors
  - Default constructor – has no arguments
  - Parameterized constructor –has arguments(parameters)

Constructor(contd.)

- **Default constructor** has no arguments or parameters.

E.g.

```
class A
{
    A( )
    {
    //statements
    }
}
```

Default Constructor of class A

Default constructor(contd.)

```
class Box
{
int width ,length,height;
Box()
{
width=10;
length=10;
height=10;
}}
```

The following statement creates an object of class Box.

Box mybox1 = **new Box();**

Here **new Box( )** is calling the **Box( )** constructor**.**

Default constructor(contd.)

```
class Box
{
int width ,length,height;
Box()
{
width=10;
length=10;
height=10;
}}
```

The following statement creates an object of class Box.

Box mybox1 = **new Box();**

Here **new Box( )** is calling the **Box( )** constructor.

Default constructor(contd.)

- When we do not explicitly define a constructor for a class, then **Java creates a default constructor for the class.**

```
class Box {
    int width;
    int length;
    int height;
    Box()
    {
    System.out.println("Constructing Box");
    width = 10;
    length = 10;
    height= 10;
    }
    int volume()
    {
    return width * length * height;
    }
}
```

```
class Box {
    int length;
    int height;
    int width;
    Box()
    {
    System.out.println("Constructor");
    width = 10;
    length = 10;
    height= 10;
    }
    int volume()
    {
    return width * length * height;
    }
}
```

```
class BoxDemo {
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
int vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

OUTPUT

Constructor
Constructor
Volume is 1000
Volume is 1000

Parameterized Constructors

- Constructors with arguments are called parameterized constructors.

```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{
width = w;
height = h;
length= l;
}
double volume()
{
return width * height * length;
}
}
```

Parameterized Constructor of class Box (Box constructor has arguments-> parameters)

```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{
width = w;
height = h;
length = l;
}

double volume()
{
return width * height * length;
}
}
```



```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
double vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
 }
}
```

```
OUTPUT
Volume is 3000
Volume is 36
```

Parameterized constructor(contd.)

Box mybox1 = new **Box(**10, 20, 15);

- Here the values 10, 20, and 15 are passed to the **Box( )** constructor when new creates the object mybox1.
- The parameterized constructor is

```
Box(double w, double h, double l)
{
width = w;
height = h;
length = l;
}
```

- Thus, value of mybox1 object's width, height, and depth will be set as 10, 20, and 15 respectively.

The this Keyword

- The **this** keyword can be used inside any method to refer to the **current object**.
- **this** is always a reference to the object on which the method was invoked.
- **this** can be used to refer current class instance variable.
- **this** can be used to invoke current class method (implicitly)
- **this**() can be used to invoke current class constructor.
- **this** can be passed as an argument in the method call.
- **this** can be passed as argument in the constructor call.
- **static methods** cannot refer to **this.**

this-Example

```
Box(double w, double h, double l)
 {
this.width = w;
this.height = h;
this.length = l;
}
```

Here **this** will always refer to the object invoking the method

```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
}
}
```

Here in statement
Box mybox1 = new Box(10, 20, 15);
mybox1 object is created by calling parameterized constructor.
Box(double w, double l, double d)
Here **this** inside constructor refers to object mybox1.

Next when **mybox2** object is created, **this** refers to object mybox2.

Instance variable hiding-using this

- We can have **local variables**, including formal parameters to methods, which has the same name of the class' **instance variables(attributes)**.
- But when a local variable has the same name as an instance variable, **the local variable *hides the* instance variable.**
  - **this** helps to solve this. Use **this.** along with instance variables.

Instance variable hiding-using this (contd.)

- // Use **this** to resolve name-space collisions.

```
class Box
{
double width
double length;
double height;
Box(double width, double height, double length)
{
this.width = width;
this.length = length;
this. height; = length;
} }
```

INSTANCE VARIABLE

CONSTRUCTOR

Instance variable hiding-using this (contd.)

- // Use **this** to resolve name-space collisions.

```
class Box
{
double width
double length;
double height
Box(double width, double height, double length)
{
this.width = width;
this.length = length;
this. height = length;
} }
```

Local variable

INSTANCE VARIABLE

Method Overloading

- It is possible to define **two or more methods** with **same name** within the same class, but their parameter declarations should be different.
  - This is called **method overloading.**
  - This is a form of **polymorphism** (many forms)
- Overloaded methods must **differ in the type and/or number of their parameters.** (return types is not significant.)
- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.

```
// Demonstrate method
    overloading.
class Over
{
void test()
{
    System.out.println("Empty");
}
void test(int a) {
    System.out.println("a: " + a);
}
void test(int a, int b) {
    System.out.println("a="+a);
    System.out.println("b="+b);
    }
}
```

```
class Sample {
public static void main(String args[])
{
Over ob = new Over();

ob.test();
ob.test(10);
ob.test(2, 5);

}
}
```

OUTPUT
Empty
a=10
a=2
a=5

- *In the example* ,test( ) is overloaded three times.
  - The first version test() takes no parameters,
  - the second **test(int a)**takes one integer parameter
  - the thrd **test(int a,int b)** takes two integer parameters.

Method Overloading(contd.)

- When an overloaded method is called**, Java looks for a match between the arguments** used to call the method and the method's parameters
- This **match need not always be exact**.
  - In some cases, Java's automatic type conversions can play a role in overload resolution.

Overloading -through automatic type conversions

```
class Over{
void test() {
System.out.println("Empty");
}

void test(double a)
 {
System.out.println("a: " + a);
}
}
```

```
class Sample {
public static void main(String
    args[])
{
Over ob = new Over();

ob.test();
ob.test(10);
ob.test(2.5);

}
}
```

OUTPUT
Empty
a=10
a=2.5

Overloading -through automatic type conversions(contd.)

- In this example when **test( ) is called with an integer argument** inside .
  - Overload, no matching method is found with int as argument**.**
- However, Java can automatically **convert an integer into a double,** and this conversion can be used to resolve the call.
  - Therefore, when **test(int) is not found,** Java elevates int to double and then **calls test(double).**

Overloading Constructors

- Constructors can be overloaded. Because a class can have any number of constructors
  - one default constructor, many parameterized constructors

```
class A
{
A() { //statements}
A(int a) { //statements}
A(int a,float b) { //statements}


}
```

```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
Box()
{
width = 0;
length =0;
height =0;
} }
```

```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box(3, 6, 2);
System.out.println("mybox1");
System.out.println(mybox1 .width + " “
    +mybox1 .length + " "+  mybox1 .height);

System.out.println("mybox2");
System.out.println(mybox2.width + " “ +
    mybox2.length + " " +  mybox2 .height);
} }
```

OUTPUT

```
mybox1
0.0   0.0   0.0
mybox2
3.0   6.0   2.0
```

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(); //ERROR
Box mybox2 = new Box(3, 6, 2);
}
}
```

ERROR

Here following statement tries to create object mybox1 of class Box ,
Box mybox1 = new Box();
This should call default constructor **Box()** in class Box.
But Box class has constructor but no default constructor is there.

So ERROR occurs

```
class Box
{
double width
double length;
double height;
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box();
}
}
```

NO ERROR in this code
The following statement creates object of Box class mybox1
Box mybox1 = new Box();
Since no constructors are not there,
Java provides the default constructor.

Using Objects as Parameters

- We can pas objects as arguments(parameters) to function(method).
- Objects are **passed by reference(call by refernce).**

Object as parameters

```java
class Test {
    int a, b;
  Test(int i, int j)
   {
   a = i;
   b = j;
   }
boolean equals(Test o)
 {
if(o.a == a && o.b == b)
 return true;
else return false;
}
}
```

```java
class PassOb {
public static void main(String args[])
{
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println(ob1.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

OUTPUT
true
false

Object as parameters

```
class Test {
int a, b;
Test(int i, int j)
{
a = i;
b = j;
}
boolean equals(Test o)
 {
if(o.a == this.a && o.b == this.b)
 return true;
else return false;
}
}
```

```
class PassOb {
public static void main(String args[])
{
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println(ob1.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

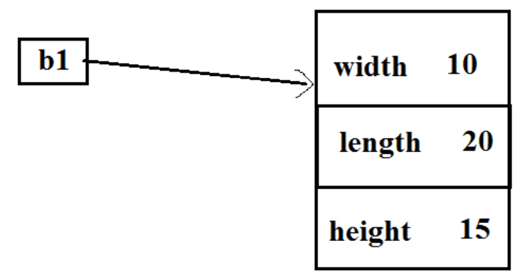
OUTPUT

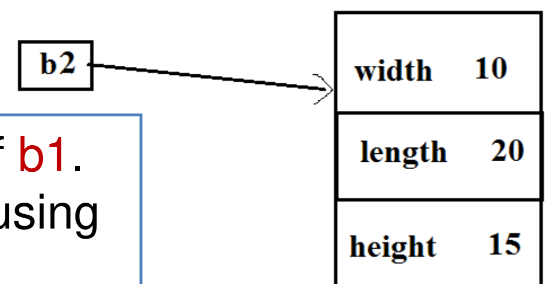
```
true
false
```

Object to initialize another object

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[])
{
Box b1 = new Box(10, 20, 15);
Box b2 = new Box(b1);
}
}
```





Here object b2 is a clone of b1. The object b2 is initialized using initial values of object b1

Passing arguments to function

- // Primitive types(int,char,double etc.) are passed by value.
- // Objects are passed by reference.

```
class Test {
int a;
Test(int i)
{
a = i;
}
void calc(Test o)
{
o.a *= 2;
}
void calc(int a)
{
a*=2;
}
}
```

```
class Obcall {
public static void main(String args[])
{
Test ob = new Test(15);
System.out.println("Object parameter");
System.out.println("Before call: " + ob.a );
ob.calc(ob);    // //Call by reference
System.out.println("After call: " + ob.a );

int a=15;
System.out.println("Integer parameter");
System.out.println("Before call: " + a);
ob.calc(a);      //Call by value
System.out.println("After call: " + a);  } }
```

```
OUTPUT
Object parameter
Before call: 15
After call: 30
Integer parameter
Before call: 15
After call: 15
```

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 7)**

Prepared by

Renetha J.B.
AP
Dept.of CSE,
Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals:

✓ **Returning Objects**

✓ **Recursion**

✓ **Access Control**

✓ **Static Members**

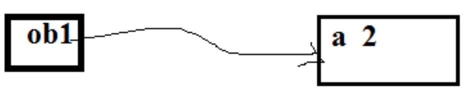
Returning objects

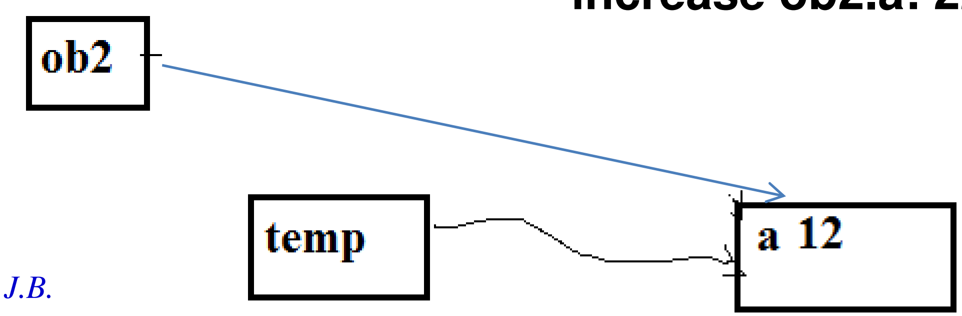
- A method can **return any type of data,**
  - **Primitive data (int ,float, char, double etc.)**
  - class types(objects) that you create.
  - etc.

```
// Returning an object.
class Test {
    int a;
    Test(int i)
    {
        a = i;
    }
Test increase()
{
Test temp = new Test(a+10);
return temp;
}
```

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.increase();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.increase ();
System.out.println("increase ob2.a: " +ob2.a);
  }
}
```

OUTPUT
ob1.a: 2
ob2.a: 12
increase ob2.a: 22





Recursion

- Recursion is the process of **defining something in terms of itself.**
- A method that calls itself is called *recursive function.*

```
// A simple example of recursion.
class Factorial {
    int fact(int n)
     {
        int result;
        if(n==1)
          return 1;
        result = n* fact(n-1) ;
        return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
int s= f.fact(3)
System.out.println("Factorial of 3 is " + s);
}
}
```

Example

```
class RecTest {
int values[];
RecTest(int i) {
values = new int[i];
}
void printArray(int i) {
if(i==0)
     return;
else
    printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
```

```
class Recursion2 {
public static void main(String args[])
    {
RecTest ob = new RecTest(5);
int i;
for(i=0; i<5; i++)
    ob.values[i] = i;
ob.printArray(5);
}
}
```

- printArray(5)
  - printArray(4)
    - printArray(3)
      - printArray(2)
        - printArray(1)
        - printArray(0)

i = 5 System.out.println("[" + (i-1) + "] " + values[i-1]); [4] 4

i = 4 System.out.println("[" + (i-1) + "] " + values[i-1]); [3] 3

i = 3 System.out.println("[" + (i-1) + "] " + values[i-1]); [2] 2

i = 2 System.out.println("[" + (i-1) + "] " + values[i-1]); [1] 1

i = 1 System.out.println("[" + (i-1) + "] " + values[i-1]); [0] 0

OUTPUT
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5

Access Control

- Through encapsulation, we can **control** what parts of a program can **access the members** of a class.
  – By controlling access, you can prevent misuse.
- How a member can be accessed is determined by the *access specifier that modifies its declaration*
- Java's access specifiers are
  - ✓ **public**
  - ✓ **private**
  - ✓ **protected**
  - ✓ ***default***

Access Control(contd.)

- When a **member** of a class is modified by the **public** specifier, then that member can be accessed by any other code. (ACCESSIBLE TO ALL)
  - public int i;
- When a member of a class is specified as **private**, then that member can only be accessed by any members of the same class.
  - private int a;

Access Control(contd.)

- When a member of a class is specified as **protected** , then that member can be accessed within the package and by any of its subclasses.

  protected char c;

- When no access specifier is there, then its access specifier is **default.**
  - It can be accessed within its own package, but **cannot be accessed outside of its package**

int c;

Access sprcifier-E.g.

```
class A{
    public int i;
    private double j;
    protected char c;
    float f;                    //default access
    public int myMethod(int a, char b)          //public method
    {   //..
    }
}
```

	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

SAME CLASS

SAME PACKAGE,

SAME PACKAGE , ANY SUBCLASS

ALL

```
class Test
{
int a;              // default access
public int b;       // public access
private int c;      // private access

void setc(int i)            //setter
{
c = i;
}

int getc()          //getter
{
return c;
}   }
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
ob.a = 10;
ob.b = 20;
// ob.c = 100; // Error! // PRIVATE
// You must access private variable c
   //through its methods
ob.setc(100);         // OK
System.out.println("a="+ ob.a);
System.out.println("b="ob.b");
System.out.println("c= " + ob.getc() );
}
}
```

static Members

- Usually we access the member of another class using object. *Syntax is:* objectname.member**;**
- If we want to access a member of another class without using object, then we have to make it a **make it a static member.**
  - Static class member is **independent of any object** of that class. We can make a member static by preceding the member declaration with the keyword **static.**

static datatype member**;**

static Members(contd.)

- When a member is declared **static,** it can be accessed before any objects of its class are created, and without reference to any object.
- Static member can be accessed using

```
classname.member;
```

static Members(contd.)

- The most common example of a **static member is** main function.
  - main( ) is declared as static because it must be called before any objects is created.
- Instance variables declared as **static** are <u>global variables</u>.
- When objects of its class are declared, separate copy of a static variable is NOT made.
- All instances(objects) of the class **share the same static variable.**

static Members(contd.)

- Methods declared as **static(static methods)** have several restrictions**:**
  - **static methods** can only call other **static methods.**
  - **static methods** must only access **static data.**
  - **static methods** cannot refer to **this** or **super.**

static Members(contd.)

- If we need to do computation to initialize your static variables, we can **declare a static block** that gets executed exactly once, when the class is first loaded.

- // Demonstrate static variables, methods, and blocks.

```
class UseStatic {
static int a = 3;
static int b;
static void show(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[])
{show(42);
}
}
```

OUTPUT
Static block initialized.
x = 42
a = 3
b = 12

Working of e.g. code

- As soon as the **UseStatic** class is loaded, all of the static statements are run.
  - First, static member **a** is set to 3,
  - then the static block executes, which prints a message and then initializes b to a * 4 or 12.
  - Then main( ) is called, which calls show( ), passing 42 to x.
  - The three println( ) statements in show refer to the two static variables a and b, as well as to the local variable x.

- if we want to call a **static method from outside its class, we** can do so using the following general form:

classname.method( )

- *Here classname is the name of the class in which the* ***static method is declared.***

Non-static method invocation

```
class Demo {
int a = 42;
int b = 99;
void callme()
{
System.out.println("a = " + a);
}
}
class Sample {
public static void main(String args[]) {
Demo dm=new Demo ();
dm.callme();
System.out.println("b = " + dm.b);
} }
```

static method invocation

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme()
{
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[])
{
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
} }
```

Nonstatic members

```
class Demo {
int a = 42;
int b=5;
void callme()
{
System.out.println("a = " + a);
}


}
class Sample {
public static void main(String args[])
    {
Demo dm=new Demo();
dm.callme();
System.out.println("b = " + dm.b);
} }
```

static members

```
class StaticDemo {
int a = 42;
static int b = 5;
static void callme()
{
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[])
{
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
} }
```

```
class Sample
{
static int a = 0;
int b;
Sample()
{
    b=0;

}
void callme()
{

a=a+2;
b=b+2;
System.out.println("static after +2 a = " + a);
System.out.println("b after +2 = " + b);
}
}
```

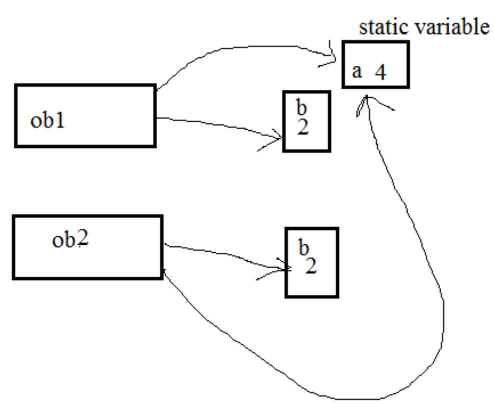
OUTPUT

ob1
static after +2 a = 2
b after +2 = 2
ob2
static after +2 a = 4
b after +2 = 2

```
class Samplestat {
public static void main(String args[])
{
Sample ob1=new Sample();
System.out.println("ob1");
ob1.callme();

Sample ob2=new Sample();
System.out.println("ob2");
ob2.callme();

} }
```



Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 8)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals**:**
  - ✓**Final Variables**
  - ✓**Inner Classes**
  - ✓**Command-Line Arguments**
  - ✓**Variable Length Arguments**

Final Variables

- A variable can be declared as **final** by prefixing **final** keyword**.**
- The contents of final variables **cannot be modified**.
- We must **initialize a final variable** when it is declared.

E.g.

final int FILE_NEW = 1;

final int FILE_OPEN = 2;

- It is a convention to choose **uppercase identifiers(CAPITAL LETTERS)** for **final variables.** E.g. TOTAL
- We can use **final variables** as if they were **constants,** without fear that a value has been changed.
- Variables declared as **final do not occupy memory on a per-instance basis.**

Nested Classes

- It is possible to define a class within another class; such classes are known as ***nested classes.***
- The scope of a nested class is bounded by the scope of its enclosing class(**outer**).
  - Thus, if class B is defined within class A, then B does not exist independently of A.

Nested Classes(contd.)

- A **nested class** has access to the members, including private members, of the **enclosing(outer) class.**
- The **enclosing class** does not have access to the members of the nested class.

Inner Classes(contd)

- A nested class, that is **declared** directly within its enclosing class scope, is a <u>member</u> of its enclosing class.

```
class Outer
{
//variables and methods
   class Inner
   {
//variables and methods
   }
}
```

- There are two types of nested classes: ***static*** *and* ***non-static***.

Inner Classes(contd)

- **Static** nested class
  - A static nested class is one that has the **static modifier** applied**.**
  - It must access the members of its enclosing class through an object.
  - It **cannot refer** to members of its enclosing class **directly**.

- // Demonstrate a STATIC inner class.

```
class Outer
 {
int outer_x = 100;
void test() {
            Nested nested= new Nested ();
            nested.display();
            }
    static class Nested {             //static nested class
            void display() {
            Outer obj = new Outer();
            System.out.println("display: outer_x = " + obj.outer_x);
                                    }
                        }
}
class NestedClassDemo {
    public static void main(String args[]) {
            Outer outer = new Outer();
            outer.test();
    } }
```

OUTPUT
display: outer_x = 100

Inner Class

- **Non static class**
  - A non-static nested class is called **inner class**.
  - An **inner class** has access to all of the variables and **methods of its outer class.**
  - It may **refer** to members of its enclosing class **directly** in the same way that other non-static members of the outer class do.

- // Demonstrate a NONSTATIC inner class.

```
class Outer
 {
int outer_x = 100;
void test() {
          Inner inner = new Inner();
          inner.display();
          }
    class Inner {
          void display() {
                    System.out.println("display: outer_x = " + outer_x);
                              }
                    }
}
class InnerClassDemo {
    public static void main(String args[]) {
          Outer outer = new Outer();
          outer.test();
    }
}
```

OUTPUT
display: outer_x = 100

- In the program, an inner class named **Inner** is defined within the scope of **class Outer.**
- Therefore, any code in **class Inner <u>can directly access</u> the** variable outer_x in Outer class**.**
- An instance method named display( ) is defined inside Inner.
  - This method displays outer_x on the standard output stream.
- The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method.
- That method creates an instance of class Inner and the display() method is called.

Inner class(contd.)

- An **instance(object) of Inner can be created only <u>within the scope of class Outer</u>.**

- We can <u>create an **instance** of Inner class **outside of Outer class**</u> by qualifying its name with Outer classname, as in

  Outer.Inner ob=outerobject.new Inner();

Inner class(contd.)

- An inner class can **access** all of the **members of its enclosing class**, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

We can define a nested class within the block defined by a method or even within the body of a **for loop**

```
// Define an inner class within a for loop.
class Outer {
int outer_x = 100;
void test() {
    for(int i=0; i<5; i++)
        {       class Inner {
                void display() {
                System.out.println("display: outer_x = " + outer_x);
                        }
                    }
            Inner inner = new Inner();
            inner.display();
                }
        }   }
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();  } }
```

OUTPUT

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Command-Line Arguments

- If we want to pass information into a program when you run it, then you can do this by passing *command-line arguments to* ***main( ).***
- A command-line argument is the information that <u>follows program's name</u> on the command line when it is executed.
- Command-line arguments are stored as strings in a **String** array passed to the args parameter of main( ).
  - The first command-line argument is stored **at args[0]**
  - the second at args[1]
  - so on.

```java
// Display all command-line arguments.
class CommandLine {
        public static void main(String args[]) {
                for(int i=0; i<args.length; i++)
                        System.out.println("args[" + i + "]: " + args[i]);
}
}
```

- Compile this usig javac and execute this program as:-

java **CommandLine** this is a test 100 -1

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Variable length arguments

- In Java methods can take a variable number of arguments.
  - This feature is called **varargs** or **variable-length arguments**.
- A method that takes a variable number of arguments is called a **variable-arity method**, or simply a **varargs** method.

Variable length arguments(contd.)

- E.g. A method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. Here it is better to pass only the arguments to which the defaults did not apply.
- E.g. printf() method can have any number of arguments.

Handling variable length arguments

- If the *maximum number of arguments is* ***small*** and ***known***, then we can create **overloaded** versions of the method, one for each way the method could be called.
- If the *maximum number of potential arguments* is ***larger***, or ***unknowable***, then the arguments can be put into an **array**, and then the array can be passed to the method.

```
class PassArray {
   static void test(int v[])
     {
    System.out.print("Number of args: " + v.length + " Contents: ");
    for(int x : v)
           System.out.print(x + " ");
    System.out.println();
    }
   public static void main(String args[])
   {
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };
    test(n1);         // 1 arg
    test(n2);         // 3 args
    test(n3);         // no args
    }
}
```

OUTPUT

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

This old method requires that these arguments be manually packaged into an array prior to calling the function test( ).

Handling variable length arguments(contd.)

- A variable-length argument is specified by three periods **(...).**
- **E.g.**

static void test(int ... v) { //statemenst }

- This syntax tells the compiler that **test( )** <u>can be called with zero or more arguments.</u>

```java
class PassArray {
static void test(int ...v)
{
System.out.print("Number of args: " + v.length + " Contents: ");
for(int x : v)
  System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
   test(10);      // 1 arg
   test(1,2,3);   // 3 args
   test();        // no args
}
}
```

OUTPUT
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:

Handling variable length arguments(contd.)

- A method can have "normal" parameters along with a variable-length parameter.
- However, the variable-length parameter must be the **last parameter** declared by the method.
- E.g:

  int test(int a, int b, double c, **int ... vals**) { //statements }

VALID

- E.g.

int test(int a, int b, double c, **int ... vals**, boolean stopFlag) { // ERROR!

Overloading Vararg Methods

- We **can overload** a method that takes a variable-length argument.
- There can be many functions with same name and having different type of variable length arguments.

- // Varargs and overloading.

```
class VarArgs3
{
    static void test(int ... v)
    {
    System.out.print("test(int ...): " + "Number of args: " + v.length);
    }
    static void test(boolean ... v)
    {
    System.out.print("test(boolean ...) " +"Number of args: " + v.length);
    }
  public static void main(String args[])
  {
        test(1, 2, 3);
        test(true, false);
  }
}
```

OUTPUT

test(int ...): Number of args: 3
test(boolean ...): Number of args: 2

Varargs and Ambiguity

- It is possible to create an ambiguous call to an overloaded varargs method.

```
class VarArgs3
{
    static void test(int ... v)
    {
    System.out.print("test(int ...): " + "Number of args: " + v.length);
    }
    static void test(boolean ... v)
    {
    System.out.print("test(boolean ...) " +"Number of args: " + v.length);
    }
  public static void main(String args[])
  {
          test(1, 2, 3);
          test();    // Error: Ambiguous!
  }
}
```

test() can call
test(int ...) or **test(boolean ...).**
Because both these functions have varargs so they can accept zero arguments .
System is confused which one to call
AMBIGUITY

Varargs and Ambiguity(contd.)

- Another e.g. of ambiguous functions

static void test(int ... v) { // ... }

static void test(int n, int ... v) { // ... }

If a call **test(2);** comes, then this will create error (ambiguous)

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 9)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals**:**
  - ✓Inheritance :
    - ✓ Super class
    - ✓Sub class
    - ✓keywords ***super***
    - ✓ ***protected*** Members

Inheritance

- Inheritance helps to create hierarchical classifications.
- Using inheritance we can create a **general class(base or super class)** that defines features **common** to a set of related items.
  - This class can then be **inherited** by other, more **specific classes(subclasses).**

Inheritance(contd.)

- A subclass is a specialized version of a superclass.
- Subclass **inherits all of the instance variables and methods defined** by the superclass and adds its own, unique elements.
- To inherit a class, we have to use **extends** keyword along *with subclass definition.*

class superclass**{ //statements……}**

class subclass **extends** superclass**{ //statements……}**

```
// A simple example of inheritance.
class A
{
    int i, j;
    void showij()
    {
    System.out.println("i and j: " + i + " " + j);
    }
}
class B extends A {
int k;
void showk() {
                System.out.println("k: " + k);
                }
void sum() {
          System.out.println("i+j+k: " + (i+j+k));
               }
}
```

Here A is the superclass of B

```
class A
{
    int i, j;
void showij()
{
System.out.println(i + "  " + j);
}
}
class B extends A
 {
int k;
void showk() {
   System.out.println("k: " + k);
                        }
void sum() {
System.out.println("sum " + (i+j+k));
                    }
}
```

```
class SimpleInheritance
{
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
superOb.i = 10;
superOb.j = 20;
System.out.println("Superobj Contents ");
superOb.showij();
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("subOb contents ");
subOb.showij();
subOb.showk();
System.out.println("Sum in subOb:");
subOb.sum(); } }
```

```
Superobj Contents
10 20
subOb contents
7 8
k: 9
Sum in subOb:24
```

Prepared by Renetha J.B.

Member Access and Inheritance

- Subclass cannot access the private members in superclass.

```
class A {
int i;    // public by default
private int j; // private to A
void setj(int x) { j = x; };
}
class B extends A {
int total;
void sum()     {
total = i + j; // ERROR, j(private) is not accessible here
                    }
}
```

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

- A major advantage of inheritance is that **once you have created a superclass** that defines the attributes **common** to a set of objects, it can be used <u>to create any number of more specific subclasses</u>.
- Each subclass can have its own special features also.

A Superclass Variable Can Reference a Subclass Object

- A **reference variable of a superclass** can be assigned a reference to any subclass derived from that superclass.

```
class A
{
}
class B extends A
{
}
class Sample
{A oba=new A();
B obb=new B();
oba=obb; }
```

Superclassobject=subclassobject

When a reference to a subclass object is assigned to a superclass reference variable, we will have **access only to those parts of the object defined by the superclass**..

```java
class Sup
{
int a,b;
void area()
{
System.out.println("Product="+ a*b);
}

}
class Sub extends Sup
{
int i;
Sub(int x,int y,int z)
{
a=x;
b=y;
i=z;
}

}
```

```java
class InhRefsub{

public static void main(String args[])
  {
  Sup supob=new Sup();
  supob.area();
  Sub subob=new Sub(10,20,30);
  supob=subob;
  supob.area();
//System.out.println("i="+ supob.i);//ERROR
  }
}
```

OUTPUT
Product=0
Product=200

Program explanaion

- Here the statement **Sup supob=new Sup();** creates an object of class Sup named supob using default constructor **Sup().** Supob has variables a and b. Since default constructor is not there, compiler provides default constructor by initializing all variables to zero, so a and b are initially 0.
- Next **supob.area();** will call area() in Sup and prints *Product=0*
- **Sub subob=new Sub(10,20,30);** creates object of Sub named **subob** using parameterized constructed ***Sub(int x,int y,int z)*** . Since Sup is the subclass of Sub, so Sub has variables a,b from Sup and i (own variable) and set a=10 b=20 i=30
- The statement **supob=subob;** assigns object **subob** to superclass object reference **supob.** So supob has value of a and b(superclass variables) same as subob. a=10 b=20
  supob.area(); will print *Product=200*

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can be done using the keyword **super.**
- **super** has two general forms**.**
  1. To call **the superclass' constructor.**
  2. To **access a member of the superclass** that has been hidden by a member of a subclass.

❑ The **static methods** cannot refer to **super.**

Using super to Call Superclass Constructors

- A **<u>subclass</u>** <u>can</u> **<u>call a constructor defined by its superclass</u>** by use of the following form of **super:**

super(*arg-list);*

- Here, *arg-list specifies any arguments needed by the constructor in the superclass*.
- ***super( )*** must always be the **first statement** executed <u>inside a subclass' constructor</u>.

```
class Sup
{
Sup()
{
System.out.println("Superclass");
}
}
class Sub extends Sup
{

Sub()
{
super();
System.out.println("Subclass");
}

}
```

```
class Supersub{

public static void main(String args[])
  {

  Sub subob=new Sub();

  }
}
```

OUTPUT
Superclass
Subclass

super keyword to acess member

- **super** always refers to the superclass of the subclass in which it is used.
- To access the member in superclass from subclass

  super.*member*

  – *Here member can be either a method or an instance variable.*
- If subclass contains same variable as superclass, then in subclass, the superclass member will be hidden by corresponding subclass member.
  – This can be prevented using super keyword

```java
class A {
int i;
}

class B extends A
{
int i;            // this i hides the i in A

B(int a, int b) {
super.i = a;      // i in A
i = b;            // i in B
}

void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
```

```java
class UseSuper {
public static void main(String args[])
{
B subOb = new B(1, 2);
subOb.show();
}
}
```

OUTPUT
i in superclass: 1
i in subclass: 2

Creating multiple hierarchy

```
class A
{
int x;
A(int p)
{
System.out.println("Superclass A ");
x=p;
}
}
class B extends A
{
int y;
B(int p,int q)
{
super(p);
System.out.println("B Subclass of A");
y=q;
}

}
```

```
Superclass A
B Subclass of A
C Subclass of A
x=10
y=20
z=10
```

```
class C extends B
{
int z;
C(int p,int q,int r)
{
super(p,q);
System.out.println("C Subclass of A");
z=r;
}

}


class Mulinh{

public static void main(String args[])
    {

    C ob=new C(10,20,30);
     System.out.println("x="+ob.x);
     System.out.println("y="+ob.y);
     System.out.println("z="+ob.x);
          }  }
```

Protected members

- Protected members are declared by prefixing the access specifierc protected.

  protected datatype member;

- The protected member in a class can be accessed by
  – any class within the **same** package.
  – direct **sub-classes in other package** also.

Protected members(contd;)

- If you want to allow an element(member) to be **seen** outside your current package, but only to classes that subclass your class directly, then declare that element (member) **protected.**

- **Eg.**

```
class  A
{
protected int c;                    //protected variable
int a;
private char b;
public float f;
protected void add()                //protected method
{  //statements
}
//methods and statements
}
```

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 10)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals**:**
  - ✓Inheritance :
    - ✓ Calling Order of Constructors
    - ✓ Method Overriding
    - ✓The Object class

Calling Order of Constructors

- Constructors are called in the order of derivation, **from superclass to subclass**
- When subclass object is created**, it first calls superclass constructor** then only it calls subclass constructor.
- If **super( ) is not used to call superclass constructor,** then the **default constructor of each superclass will be executed** before executing subclass constructors.

```
class A
{
int i;

A()
{
 System.out.println("Constructor of superclass A");
}
}
class B extends A
{
int j;
B()
{
  System.out.println("Constructor of subclass B");
}
}
```

```
class Consorder
{

 public static void main(String args[])
 {
  B obb =new B();
 }

}
```

OUTPUT
Constructor of superclass A
Constructor of subclass B

```
class A
{
int i;
A()
{
System.out.println("Constructor of superclass A");
}
}
class B extends A
{
int j;
B()
{
System.out.println("Constructor of subclass B");
}
}
```

```
class C extends B
{
int j;
C()
{
System.out.println("Constructor of subclass C");
}
}
class Consorder{

public static void main(String args[])
{
C obc =new C();
}

}
```

OUTPUT
Constructor of superclass A
Constructor of subclass B
Constructor of subclass C

Calling Order of Constructors(contd.)

- Superclass has no knowledge of any subclass, any initialization it needs to perform is separate and it should be done as a prerequisite to initialize the subclass object.
- Therefore, **superclass constructors are executed** before executing subclass constructors, <u>when we create subclass object.</u>

Method Overriding

- In a class hierarchy, when a **method in a subclass** has the same name and type signature as a **method in its superclass**, then the *method in the* ***subclass*** *is said to* ***override the method in the superclass.***
- This is called METHOD OVERRIDING

Method Overriding(contd.)

- When an overridden method is called from within a subclass, it will always ***refer to the method defined by the subclass***.
  - The version of *the method defined by the superclass will be hidden.*

- // Method overriding.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println(" i : " + i + " j: " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
System.out.println("k: " + k);
}
}
```

```
class Override {
public static void main(String args[])
{
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

OUTPUT
k: 3

When **show( )** is invoked on an object of type B, the version of **show( ) defined within B is used**.
That is, the version of show( ) inside subclass B overrides the version declared in superclass A.

- // No method overriding.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println(" i : " + i + " j: " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void display() {
System.out.println("k: " + k);
}
}
```

```
class Sample{
public static void main(String args[])
{
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in A
}
}
```

OUTPUT
i: 1 j: 2

Here when **show( )** is invoked on an object of type B, since the version of **show( ) is not defined within B** the version of show() declared in superclass A is called and excuted.

Method Overriding(contd.)

- To access the **superclass version of an overridden method**, we can do using **super** keyword**.**

- // Method overriding.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println(" i : " + i + " j: " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show();
System.out.println("k: " + k);
}
}
```

```
class Override {
public static void main(String args[])
{
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

OUTPUT
i:1 j:2
k: 3

When **show( )** is invoked on an object of type B, the version of **show( ) defined within B is used**..

super.show() calls the show() method in its superclas.

Method Overriding(contd.)

- **Method overriding** occurs only when the ***names and the type signatures of the methods*** in subclass and superclass are **identical**.
- If ***names and the type signatures of the two*** **methods are different**, then the two methods are simply **overloaded.**

```
class A {
int i, j;
A() {
i = 0;
j = 0;
}
void show() {
System.out.println(show in A);
}
}
class B extends A {
int k;
B() {
k = 0;
}
void show(String msg) {
System.out.println("show in subclass B");
}
}
```

```
class Sample {
public static void main(String args[]) {
B subOb = new B();
subOb.show("k is "); // this calls show() in B
subOb.show(); // this calls show() in A
}
```

```
show in subclass B
show in A
```

Here show() Methods have differing type signatures. So they are overloaded **– not overridden**

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i : " + i + "j: " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show(String msg) {
System.out.println(msg + k);
}
}
```

```
class Sample {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("k is ");
subOb.show();
}
```

Here show() Methods have differing type signatures. So they are overloaded **– not overridden**

Object

- There is one special class, **Object**, defined by Java.
- All other classes are subclasses of **Object.**
- That is, **Object is a superclass of all other classes.**
- Reference variable of type **Object** can refer to an object of any other class.

Methods in Object class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Methods in Object class(contd.)

- The methods **getClass( ), notify( ), notifyAll( ), and wait( )** are declared as final.
- The **equals( )** method **compares the** contents of two objects.
  - It returns **true if the objects are equivalent, and false otherwise.**
- The **toString( )** method **returns a string** that contains a description of the object on which it is called.
  - This method is automatically called when an object is output using **println( ).**
  - **Many classes override this method.**

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 11)**

Prepared by

Renetha J.B.
AP
Dept.of CSE,
Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals**:**
  - ✓Inheritance :
    - ✓Abstract Classes and Methods
    - ✓using ***final*** with Inheritance.

Abstract Classes and Methods

- Sometimes we may want to create a **superclass** that only defines a generalized form which will be shared by all of its subclasses and leaves the implementation to be filled by each subclass.
  - To ensure that **a subclass should override all necessary methods(implementations),** we have to make them **abstract methods in superclass.**
- For making a method an **abstract method** we have use **abstract** type modifier.

Abstract Classes and Methods(contd.)

- Abstract methods have **no implementation(function body) in the superclass** .
  - so they are also called as ***subclasser responsibility***
  - the **implementation** should be there **in subclasses by overriding** those methods.
- To declare an **abstract method in <u>superclass</u>,** syntax is :

abstract *type name(parameter-list);*

- The semicolon ; after the function header shows that abstract function has no body in superclass.

.

Abstract Classes and Methods(contd.)

- **ABSTRACT CLASS**
  - Any **class** that contains **<u>one or more abstract methods</u>** must also be declared abstract.
  - To declare a class abstract, use the **abstract** keyword in front of the class keyword at the beginning of the class declaration.

```
abstract class classname
{
//members.abstract or nonabsract method
}
```

- Abstract class can have <u>non abstract methods</u>(**concrete** methods) also.

Abstract Classes and Methods(contd.)

- Abstract classes cannot be instantiated using **new** operator.
  - i.e. **Objects are not created** from abstract class.
  - Such objects would be ***useless***, because an abstract class is not fully defined.
- There are **no** abstract constructors, or no abstract static methods.
- Any **subclass** of an **abstract class** must **either *implement all of the abstract methods*** in the superclass, ***or*** it should be ***declared abstract class*.**

// A Simple demonstration of abstract with abstract and concrete methods.

```
abstract class A
{
abstract void callme();

void callmetoo()
{
System.out.println("concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("callme in B");
                    }
}
```

```
class AbstractDemo {
public static void main(String args[])
    {
     B b = new B();
     b.callme();
     b.callmetoo();
    }
}
```

OUTPUT
callme in B
concrete method.

Abstract Classes(contd.)

- Although abstract classes cannot be used to instantiate objects, abstract classes can be used to **create object references**,

```
superlassname  superclassobjectrefernce ;
superclassobjectrefernce =subclassobjectreference;
```

- Java's ***run-time polymorphism(dynamic binding)*** ***is implemented through the use of superclass references.***

// DYNAMIC(run-time) BINDING(polymorphism).

```
abstract class Figure
{   double dim1;
    double dim2;
    Figure(double a, double b)
    { dim1 = a;
    dim2 = b;
    }
    abstract double area();
}
class Rectangle extends Figure
{
Rectangle(double a, double b)
{  super(a, b);
}
double area()
{   System.out.println("Rectangle Area");
    return dim1 * dim2;
} }
class Triangle extends Figure {
Triangle(double a, double b)
{ super(a, b);
}
double area()
{ System.out.println("Triangle Area");
return dim1 * dim2 / 2;
} }
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;  // superclass object reference
figref = r;     //figref  refers to object of Rectangle
System.out.println("Area is " + figref.area());
figref = t; //figref  refers to object of Triangle
System.out.println("Area is " + figref.area());
}
}
```

Dynamic binding(program)contd.

OUTPUT

```
Rectangle Area
Area is 45.0
Triangle Area
Area is 40.0
```

- Here all subclasses of abstract class **Figure** must override abstract method area( ).
- Here the statement **Figure figref;** is notcreating object but creating an object refernce.
- The statement **figref = r;** means that superclass reference figref now points to subclass(Rectangle) object r. So the value of dim1 and dim2 are the values in r. The statement **figref.area()** will call area() method in that subclass(Rectangle)
- The statement **figref = t;** means that superclass reference figref now points to subclass(Triangle) object t. So the value of dim1 and dim2 are the values in t. The statement **figref.area()** will call area() method in that subclas(Triangle)

Using final with Inheritance

- Use of **final** keyword
  - **final** can be used to create the equivalent of a named constant(***final variable***). E.g. final int TOTAL=0;
  - **final** helps to prevent overriding in inheritance
  - **final** helps to prevent inheritance.

Using final with Inheritance

- **Using final to Prevent Overriding**
  - If we don't want to allow subclass to override a method of supeclasses, we can use **final** as a modifier at the start of its method declaration in superclass.
  - Methods declared as **final** *cannot be overridden by subclass.*

Using final to Prevent Overriding(contd.)

```
class A {
    final void show()
    {  System.out.println("This is a final method.");
    }
}
class B extends A {
    void show()                    // ERROR! Can't override.
    { System.out.println("Illegal!");
    }
}
```

Here show() method is declared as final in A. So it cannot be overridden(redefined) in subclass B. If we try to override, COMPILE ERROR will occur in the program.

Using final to Prevent Overriding(contd.)

- Methods declared as **final** can sometimes provide a **performance enhancement**:
  - The compiler is free **call them inline** because it "knows" they will not be overridden by a subclass.
- When a small **final** method is called, Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus *eliminating the costly overhead associated with a method call.*
- Inlining is only an option with final methods.

Using final to Prevent Overriding(contd.)

- Normally, Java resolves calls to **methods** dynamically, at run time. This is called **late binding.**
- However, since **final methods** **cannot be overridden**, a call to final method can be resolved at compile time. This is called **early binding.**

Using final to Prevent Inheritance

- To prevent a class from being inherited it can be declared as final.
  - We cannot create subclasses from a final class.
- Class with final modifier cannot be inherited. It **cannot act as superclass.**
- To make a class a final class, precede the class declaration with the modifier **final**.
- If we declare a class as **final**, it implicitly declares **all of its methods as final.**
- It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself.

Using final to Prevent Inheritance

final class A {

// ...

}

// The following class is illegal.

class B extends A { //ERROR!cannot create a subclass for final class A

// ...

}

It is illegal for B to inherit A since class **A is declared as final.**

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.